Bolt Beranek and Newman Inc.

Report No. 3803

ARPANET Routing Algorithm Improvements
First Semiannual Technical Report

Dr. J.M. McQuillan, Dr. I. Richer, Dr. E.C. Rosen

April 1978

Prepared for:
Defense Advanced Research Projects Agency
and
Defense Communications Agency

Report No. 3803 Bolt Beranek and Newman Inc.

BBN Report No. 3803

ARPANET Routing Algorithm Improvements

First Semiannual Technical Report

April 1978

SPONSORED BY
DEFENSE ADVANCED RESEARCH PROJECTS AGENCY AND
DEFENSE COMMUNICATIONS AGENCY (DOD)
MONITORED BY DSSW UNDER CONTRACT NO. MDA903-78-C-0129

ARPA Order No. 3491

Submitted to:

Director
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, VA 22209

Attention: Program Management

and to:

Defense Communications Engineering Center
1860 Wiehle Avenue
Reston, VA 22090

Attention: Dr. R.E. Lyons

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: BBN - 3803 (14)
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle): (6) ARPANET Routing Algorithm Improvements

5. TYPE OF REPORT & PERIOD COVERED: Semiannual Technical Report, 10/1/77 to 4/1/78

6. PERFORMING ORG. REPORT NUMBER: 3803

7. AUTHOR(s): (10) J.M. McQuillan, I. Richer, E.C. Rosen

8. CONTRACT OR GRANT NUMBER(s): (15) MDA903-78-C-0129, ARPA Order 3491

9. PERFORMING ORGANIZATION NAME AND ADDRESS: Bolt Beranek and Newman Inc., 50 Moulton Street, Cambridge, MA 02138

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: ARPA Order No. 3491

11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Projects Agency, 1400 Wilson Blvd., Arlington, VA 22209

12. REPORT DATE: (11) April [redacted]78

13. NUMBER OF PAGES: 192 (12) 198 p.

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Defense Supply Service - Washington, Room 1D 245, The Pentagon, Washington, D.C. 20310

15. SECURITY CLASS. (of this report): UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

~~UNCLASSIFIED/UNLIMITED~~

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

(9) Semiannual Technical Rept No. 1, 1 Oct 77 - 1 Apr 78,

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

computer networks, routing algorithms, ARPANET, protocols, congestion control, shortest path problem, line up/down procedures, network delay, network measurement, logical addressing, multi-destination addressing

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report describes progress during the first six months of a contract to make several improvements to ARPANET routing. Some principal conclusions are: Several problems have been discovered in the present congestion control, line up/down procedures, and loop suppression techniques through the use of a new real-time monitoring capability. Solutions to most of these problems have been developed; operational experience with these changes is presented. A new set of line up/down

continue --

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

060 100

(20.--continued)
procedures with much better performance was developed and analyzed. Extensive measurements of network delay were carried out; delays fluctuate rapidly over a large range, making it difficult to devise effective estimation techniques. The present routing algorithm is itself a major contributor to network delay due to the computations and update messages it requires. A set of improvements was designed for the present routing algorithm to make it more efficient and effective. However, a new algorithm, based on performing the entire shortest path calculation at each node, incrementally for each network change, appears to be a better choice for installation in the ARPANET. The new algorithm, SPF, has been programmed for the IMP, and its performance agrees with analytical results. A mechanism for multi-destination addressing in the ARPANET was designed and analyzed.

TABLE OF CONTENTS

1. OVERVIEW

This report covers work performed in the period October 15, 1977 through April 15, 1978 on the contract to study ARPANET routing algorithm improvements. Our progress to date is reported in six main sections which follow.

Some of the conclusions reached to date in the study are summarized below for each of these sections (figures in parentheses denote the pertinent section number):

Analysis of Present ARPANET Performance (2)

-- The change made to the ARPANET causing it to drop packets which have been unsuccessfully retransmitted 32 times has greatly reduced the number of network disturbances (2.1).

-- There are significant problems with the present protocols in that lines can be declared up in one direction for long periods of time (2.2).

-- We have discovered several classes of routing failures that cause packets to loop between two or more IMPs, and have identified the causes of these problems (2.3).

-- We have performed extensive measurements of network delay and discovered that it is extremely variable. This indicates that routing algorithms which measure instantaneous delay are inappropriate for the ARPANET. Furthermore, periodic routing calculations and updates in the ARPANET are contributing factors to the variability in delay (2.4).

-- A new "snapshot" measurement package has been developed as a tool for monitoring network performance, especially at the time of some transient phenomena such as topology changes, packet loops, etc. (2.5).

Line Up/Down Protocols (3)

-- A description of the existing line up/down procedures and an explanation of observed problems is given (3.2).

-- A set of goals is suggested for handling lines with acceptable quality and unacceptable quality in order to insure that variations in line quality will not degrade network performance (3.3).

-- A consecutive counter for bringing a line up and the criterion of missing 3 within the last n test messages for bringing a line down are shown to be adequate for the goals suggested (3.4).

-- These procedures do not completely eliminate the possibility of lines being declared up in one direction. A new state will be added to the line up/down protocol to remove a known cause of network congestion (3.5).

Measurement of Network Delay (4)

-- It is important to measure network delays directly rather than to estimate them indirectly by other network parameters (4.1).

-- We experimented with various means for smoothing the measured delay values. An average over the last time period is more appropriate than a median type of smoothing algorithm (4.3).

-- There is a strong need for continued measurement of delay and re-evaluation of smoothing functions after implementing any new algorithm since the behavior of network delay is strongly influenced by the type of routing algorithm employed (4.4, 4.5).

Improvements to the Current ARPANET Routing Algorithm (5)

-- The ARPANET routing algorithm can be modified to use event-driven updating which leads to significant improvements in performance. However, there are still some difficulties with this approach (5.1).

-- The current routing algorithm determines reachability very slowly and uses too much bandwidth in doing so. This can be improved somewhat but the basic problem rests with using a path-based routing algorithm relying on hop counts and timers to determine whether nodes are reachable. Since this is an essential feature of any ARPANET-like algorithm, we were led to consider other types of procedures (5.2).

-- Hold-down can be replaced with a less heuristic form of loop suppression. This could improve performance without significantly increasing cost (5.3).

A New Routing Algorithm--Shortest Path First (6)

-- We have demonstrated that it is practical to implement a separate and independent shortest path calculation in each of the IMPs in the ARPANET as opposed to the present distributed computation (6.1, 6.2).

-- Such an algorithm can be designed to be very efficient in space and time, using as little as one or two milliseconds of CPU time, on the average, to perform an individual update when the calculation is performed incrementally (6.3).

-- Efficient and reliable updating procedures can be developed so that an incremental shortest path algorithm can be performed on an event-driven basis (6.4).

-- The incremental shortest path algorithm has significant advantages over the present ARPANET algorithm in terms of efficiency, reliability, loop freedom and speed of adaptation (6.5).

-- We will continue to investigate the SPF algorithm as a candidate for eventual installation into the ARPANET (6.6).

Message Addressing Modes (7)

-- A multi-destination addressing technique is proposed for significantly reducing the number of packet hops required for transmission of a multi-destination message compared to that required for separately-addressed messages (7).

-- Installation of such a mechanism in the ARPANET will be easier for group addressed messages than for messages explicitly addressed to multiple destinations (7).

-- Multi-destination addressing and multiple homing of hosts are mechanisms which will be installed in the

ARPANET for datagram traffic but probably not for virtual circuit traffic due to the complexities involved in adapting the protocol to multiple sources and destinations (7).

2. CURRENT STATUS OF ARPANET ROUTING

This section describes the current status of routing and related topics in the ARPANET. Section 2.1 is a brief summary of the results of a two-month study of routing problems which BBN performed for DCA in the summer of 1977 (see BBN Report 3641). Sections 2.2 and 2.3 describe some problems which were noted during that study, but not resolved at that time. Some of these problems have been subsequently eliminated; others will be eliminated in the next few months. Section 2.4 describes some of the characteristics of store-and-forward delay in the ARPANET, and discusses the ways in which these delays are related to various characteristics of the current routing algorithm. Section 2.5 describes a measurement package which we developed especially for monitoring the performance of routing.

2.1 Summary of Previous Work on Routing and Congestion

In late 1976 and early 1977 the ARPANET was subject to a number of disturbances which made it virtually unusable for short periods of time. These disturbances often occurred several times a week, leading to a serious degradation in the performance of the network. By using the measurement package described in Section 2.5, we were able to determine that the disturbances had the following etiology. First, some IMP would become congested;

that is, it was often forced to refuse packets from its neighbors because it had no buffering available. Then the congestion would spread, affecting a large portion of the network. Finally, many IMPs would declare their lines to be down, causing the network to partition. This caused many IMPs to break their connections to each other. As a result, IMPs would discard packets destined for the now unreachable IMPs, and this cleared up the congestion. After several minutes, normal network operation was resumed.

Thus each disturbance had two components: the start of congestion, and the spreading of that congestion. There are many situations which may cause some IMP to be congested; two very important causes were found to be line up/down mismatches (discussed in Section 2.2) and routing loops (discussed in Section 2.3). It is easy to see why routing loops can cause congestion. Packets caught in a loop are stuck in the network for an arbitrarily long period of time, thereby wasting buffer space and reducing the network capacity. Furthermore, if packets are looping between a pair of IMPs, a sort of lock-up can occur, with each IMP filled with packets for the other. This makes the line between them useless.

The two major reasons for the spread of congestion are related to (1) the performance of the routing algorithm, and (2) the the IMP-IMP protocol. The problem with the routing algorithm

is that it is both unable to prevent congestion from arising, and unable to detect the presence of congestion once it exists. These failings have to do largely (though not exclusively) with the algorithm's method of measuring delay by taking instantaneous samples of the queue length. As is shown in Section 2.4, instantaneous samples of the queue length are poorly correlated with expected delay. In addition, the queue lengths have a relatively small dynamic range, while delay does not. Section 4 discusses better ways of measuring the delay. However, even with better delay measures, the routing algorithm would still react poorly to congestion, since it is slow to react to any change in the network. This issue is discussed further in Section 5.

The problem with the IMP-IMP protocol was that when an IMP became congested and had no space to buffer any more packets, it would refuse to acknowledge packets sent to it by its neighbors. The neighbors would try to retransmit such packets up to 600 times (over a period of 75 approximately seconds), after which the line over which they were being transmitted would be declared down. Unfortunately, these procedures only made the congestion worse. Congestion arises when the demand for buffering resources exceeds the buffer supply. When an IMP becomes congested its neighbors would dedicate buffers to a single packet for up to 75 seconds, instead of the more usual few milliseconds. This

greatly increased the demand on buffering resources in the neighbors, causing the congestion to spread to them. Furthermore, artificially declaring a good line to be unusable only serves to further deplete network resources without decreasing the demand on them.

Experimentation showed that the precise value of the retransmission limit is not too significant--altering it does change the characteristics of the disturbances, but does not eliminate them. The only effective way we found to prevent the spread of congestion was to reduce the demand on network resources by discarding packets. Currently, the network will discard any packet that is retransmited 32 times (over an interval of 4 seconds). Over the last six months, experience has shown this procedure to be successful in preventing the spread of congestion.

2.2 Line Up/Down Mismatches

It is presently possible for two IMPs which are connected by a particular line to disagree as to the state of the line, with one of the IMPs declaring it up and the other declaring it down. The presence of such a line up/down mismatch can have an extremely deleterious effect on network performance since the IMP-IMP protocol does not operate properly on a mismatched line.

The IMP which has decided that the line is up will continue to transmit packets over it, but the other IMP does not return acknowledgments for those packets. This can cause the packets to be retransmited up to the maximum number of times, after which they are discarded from the network. Thus mismatches are a major source of network congestion and packet loss.

We have observed mismatches happening frequently (several times a week) and lasting for periods as long as seven hours (though several minutes is more typical). Network performance would be significantly improved if mismatches were eliminated by changing the line up/down protocols to prevent one IMP from declaring a line up when the adjacent IMP does not. Section 3 describes a method for achieving this goal.

2.3 Loops in ARPANET Routing

This section describes looping problems that have been observed in the ARPANET. The problem described in Section 2.3.1 was resolved by a minor modification; the problems described in Sections 2.3.2 and 2.3.3 cannot be eliminated without making major changes to the routing algorithm (see Section 5).

2.3.1 Routing to Singly-Connected IMPs

We recently discovered a kind of routing loop which can cause packets to be routed to singly-connected nodes by mistake. Suppose there are two independent paths from IMP A to IMP X, and that these paths are approximately equidistant in terms of delay. Let B and C be the two neighbors of A which are the "next hops" on these two paths, respectively. Let S be a third neighbor of A which is not on a reasonable path from A to X. S may be a singly-connected node, like the IMP at Hawaii, or it may have other neighbors. What is important is that S's best path to X be via A.

Now suppose that A is routing traffic to X via B. Let d be the delay that A sees to X via B. Consider the following sequence of events:

1. At t0, A sends routing to S, reporting that the delay to X is d.

2. At t1, A receives and processes routing from B. As a result, A now sees a delay of d+5 to X (via B). Since the increase in delay is only 5, A does not hold down.

3. At t2, A receives and processes routing from C. As a result, A now sees a delay of d+4 to X via C. Since the delay via B is d+5, A switches paths, so that traffic to X is now routed via C.

4. At t3, A receives and processes routing from C again. (Note that C is sending routing more frequently than S is.) As a result, A now sees a delay of d+10 to X via C. Since the increase in delay is only 6, A does not hold down.

5. At t4, A receives (finally) and processes routing from S. Since the routing message from S is based on the last routing message that A sent to S, back when the delay to X was only d, A now sees delay to X via S of d+8. Since the delay to X via C is d+10, A switches paths, routing traffic to X via S. Since S is, ex hypothesi, routing traffic to X via A, a loop has formed.

Between the time A sends routing to S, and the time it gets routing back, the delay A sees to X has increased from d to d+10. However, at the time A receives routing from S, the increase in delay on its current best path (via C) has only increased by 6, from d+4 to d+10. While the IMP accumulates increases in delay for a period of two ticks, it only accumulates increases on its current best path. When it switches paths, it simply throws away all information about increases in delay on the previous path. As shown above, This can cause it to fail to hold down when it should, thereby permitting loops.

The solution is that an IMP should hold down when its delay to a destination increases by 8 during a period of two ticks, even if that increase was partially along one path and partially along another. This modification has been made, and this type of loop has not been observed any more.

2.3.2 The Spread of Routing Loops

Another kind of routing failure has been noted in the network. It begins when a pair of neighboring IMPs report that packets to a particular destination are looping between them. Shortly thereafter, other neighbors of this pair of IMPs report looping packets to that same destination. Then neighbors of the neighbors of the original pair of IMPs report looping packets to

that same destination. Then neighbors of the neighbors of the neighbors...etc. This phenomenon has been observed to spread quite far, with as many as 34 different IMPs reporting looping packets to the same destination. Eventually, routing stabilizes and the reports cease. This phenomenon is explained below and illustrated in Figures 2-1 and 2-2.

Let A and B be a pair of neighboring IMPs. Let X be a third IMP such that there is at least one path from A to X, and a second path from B to X which is independent of the first, and approximately equidistant in terms of delay. Whenever this is the case, it is possible for packets to X to loop between A and B. That is, it is possible that A will decide its best path to X is via B, and B will decide its best path to X is via A. Let C be A's neighbor on its path to X, and 1 D be B's neighbor on its path to X (see Figure 2-1) and consider the following sequence of events:

1. At t0, A's delay to X via C is d, and B's delay to X via D is d. A and B send routing to each other.

2. At t1, A and B get routing from C and D, respectively. As a result, A now has a delay to X via C of d+5, and B has a delay to X via D of d+5.

3. At t2, A and B receive from each other the routing that they sent to each other at t0. As a result, A now has a delay to X via B of d+4. Since its delay to X via C is d+5, A switches paths. Similarly, B now has a delay to X via A of d+4. Since its delay to X via D is d+5, B switches paths. At this point, a loop has formed between A and B.

It is obvious that when such a loop forms, A and B will enter hold-down as soon as they exchange routing. In fact, they will re-enter hold-down every time they exchange routing, since the delay each sees to X will increase by 8 with each routing update. Since A and B are bound to exchange routing at least once before the hold-down timer expires, and since they re-enter hold-down whenever they exchange routing, they will never leave hold-down. Or rather, they will never leave hold-down unless some special action is taken. And as long as the situation persists, no packet will ever be able to get from A or B to X.

To prevent this situation from persisting, the following strategy is presently used: If A receives a packet for X from B, then since A's route is via B, A simply comes out of hold-down prematurely. Of course, coming out of hold-down does not break the loop. Whether the loop gets broken depends on which of A's neighbors is the next to send it routing. There are two cases to

consider: either A's next routing update comes from B, or else it comes from some other neighbor of A (call it E). In the former case, the loop is not broken, and A just goes back into hold-down. The latter case has two sub-cases: either A switches its path to X from B to E, or it does not. Only in the former case (which is by no means the inevitable case) is the original loop broken.

Let us suppose then that A switches its best path to X from B to E, thereby breaking the loop between A and B. Now we have two more cases to consider. Either E's path to X is via A, or it is not. In the latter case, everything is fine. But in the former case, there is more trouble. Now there is a loop between A and E (see Figure 2-2) The same process could potentially be repeated indefinitely until the phenomenon spreads to every IMP in the network. Thus the phenomenon of looping packets to a given destination can spread away from the location of the original loop.

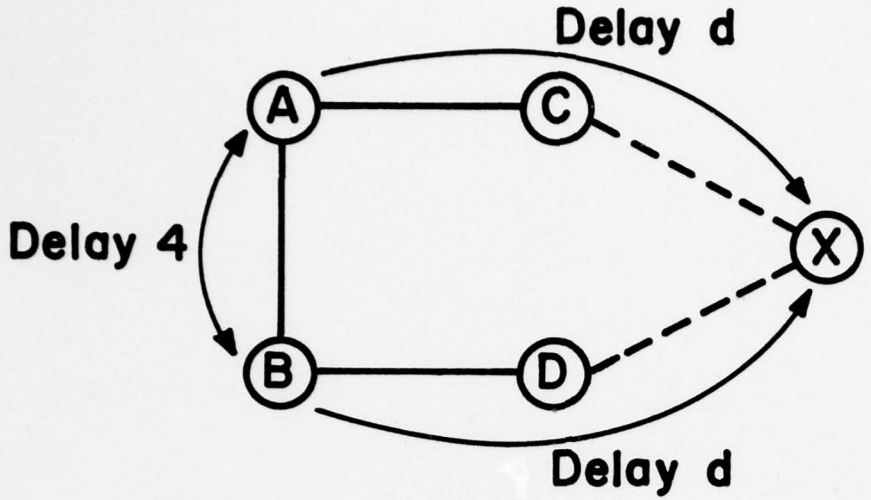
Clearly, the extent of the spread is directly proportional to the number of IMPs which are directing their traffic to X towards the area of the original loop. If there is a loop between A and B, and every other IMP in the network is directing its traffic for X to either A or B, then the phenomenon will spread far; if no other IMP is directing its traffic for X to A

or B, no spreading will occur. When a loop forms between A and B, the delay that these two nodes see to X will be constantly increasing. Therefore, most of the time when such a loop forms, most traffic gets directed away from the loop, and the loop does not spread. However, there are certain unusual conditions which can cause many other nodes in the network to direct their traffic towards the loop. For instance, suppose one or more of the lines along a real path to the destination X go down. Then, for a period of time, many nodes will see infinite delay to X. However, the two nodes between which there is a loop to X will not see infinite delay. They will see only a constantly increasing finite delay. This will cause many nodes to direct their traffic for X toward the loop, and thus will cause the loop to spread. Since this in turn causes many IMPs to come out of hold-down prematurely, the ultimate effect is that the network is forced to adjust to some "bad news"--an increase in delay--without the benefit of hold-down. As is well-known, this can take a long time.

This phenomenon has been observed frequently in the network. It usually starts in the Washington area, with the first loop either between NBS and NSA or NBS and PENT or NBS and ABER. The destination of the looping packets is usually (though not always). one or more of the European IMPs, and Europe is usually

(though not always) unreachable while the phenomenon is occurring.

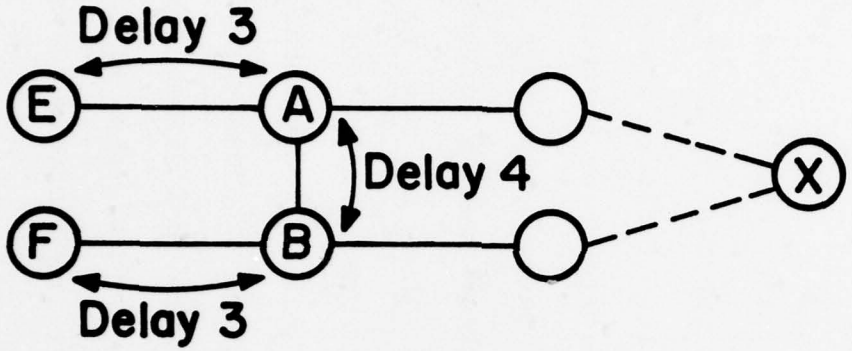
While the current routing algorithm may be more prone to forming loops than some other algorithms, it is doubtful that there can be any routing algorithm which can be guaranteed to be loop-free in actual operation. Thus some scheme for detecting and breaking loops will always be needed.



ROUTING UPDATES	DELAY A→X	DELAY B→X	COMMENT
1. A and B Send	d, via C	d, via D	Steady State
2. A and B Receive from C and D	d + 5, via C	d + 5, via D	Delay Increased
3. A and B Receive from Each Other (Sent at 1)	d + 4, via B	d + 4, via A	Loop Formed

Figure 2-1 Formation of Routing Loops

- Hold Down Causes Loops to Lock Up
- To Prevent This Loops Are Broken
- But Hold Down is Terminated Prematurely
- New Routing Loops May Form



ROUTING UPDATES	DELAY A→X	DELAY B→X	COMMENT
4. A and B Send	d+4, via B	d+4, via A	A and B in a Loop
5. A and B Receive from Each Other	d+12, via B	d+12, via A	Hold Down in a Loop
6. A and B Break Hold Down			
7. A and B Receive from E and F	d+10, via E	d+10, via F	A and E in a Loop B and F in a Loop

Figure 2-2 Spread of Routing Loops

2.3.3 Loops Due to Different Updating Rates

Routing update messages are transmitted periodically by each IMP. However, the phasing among IMPs is essentially random. As a result, the same information may travel much more rapidly along one path than along another. This can result in a certain kind of loop, as shown in Figure 2-3. Suppose that all IMPs are sending traffic to IMP X along the routes indicated in the figure. Suppose further that the line to IMP X goes down, and that the information about the line failure travels much more rapidly in the counter-clockwise direction than in the clockwise direction. Then several of the nodes on the upper part of the ring will have time to change their routing so that they send to X via A, before A determines that it has no path to X. Eventually, A will decide that it has no path, and will pass this information around clockwise. This will cause the nodes closest to A to realize they have no path to X. However, nodes farther from A have now begun to route to X via A. It is as if the correct information ("no path to X") is chasing the incorrect ("path to X via A") around in the clockwise direction, but is never able to overtake it.

While loops of this sort are certainly possible, it is not known how often they occur. No fully satisfactory way of eliminating them has been devised.

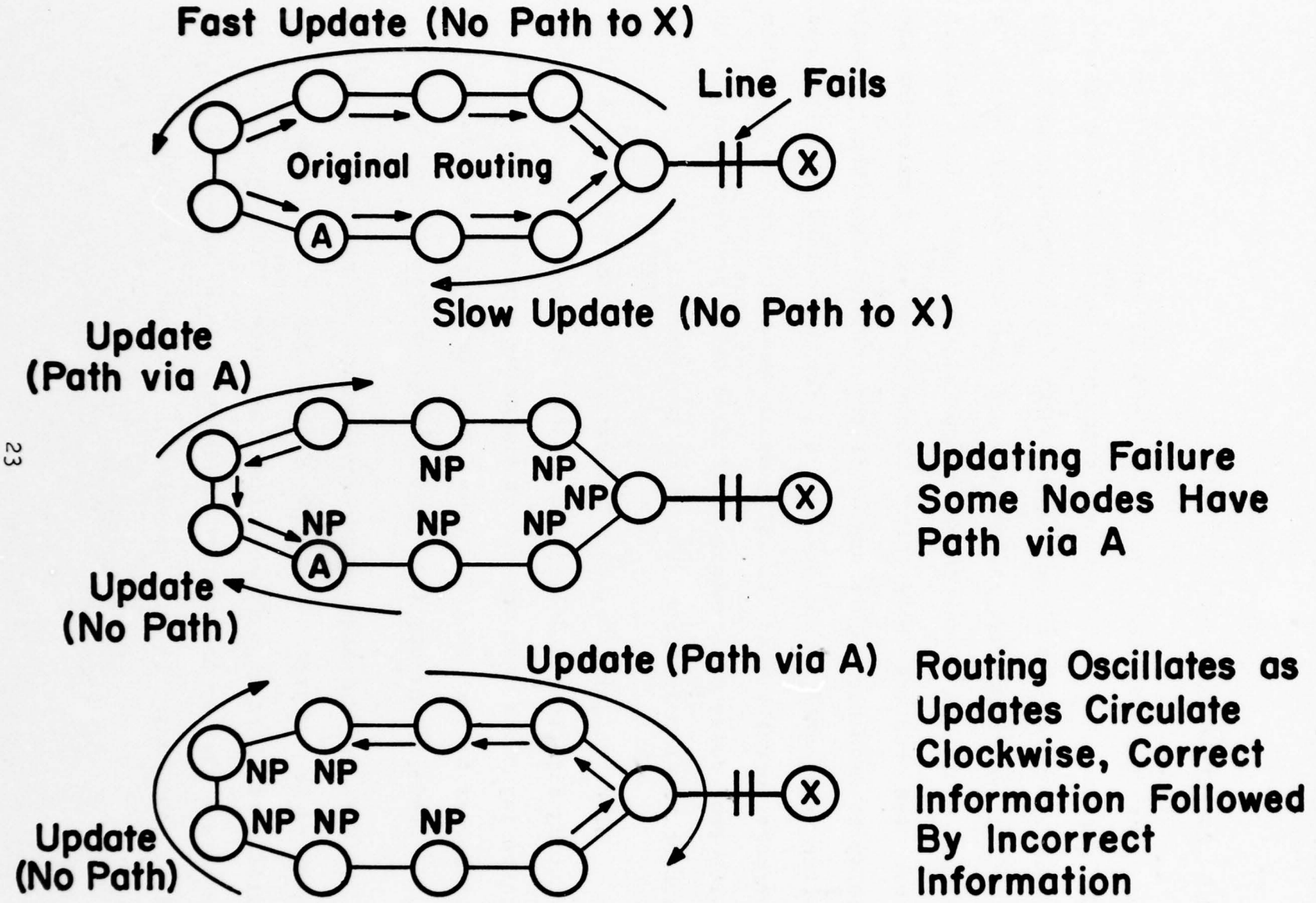


Figure 2-3 A Multi-node Routing Loop

2.4 Packet Delays in the ARPANET

One very important goal of the ARPANET routing algorithm is to ensure that packets travel over paths which minimize network delay. In order to determine the delay on a particular path, the routing algorithm must in some way add up the delays which a packet would experience on each "hop" of that path. This implies that the routing algorithm must have some way of measuring the delay at each hop. This aspect of the routing algorithm, though often neglected, is quite crucial, for no routing algorithm can be more accurate than its delay measurement portion--an algorithm with poor delay measurement facilities will perform poorly, no matter how sophisticated its other features are.

With an eye towards evaluating and improving the ARPANET routing algorithm's delay measurement facilities, we have been gathering data from the net on the delay a packet experiences as it passes through an IMP. (Our data gathering methodology is described in Appendix 1.) This section reports on that data.

When we plotted store-and-forward delay and its various components against time, we found that data gathered from different IMPs at different times and under different conditions were all similar in important respects. This leads us to believe that our data is not atypical, and can be used to draw

conclusions about store-and-forward delays in the ARPANET generally. Of course, only a small amount of the data we've gathered can be reproduced here. In all of the plots reproduced here, packet delay (in 10's of milliseconds) is on the y-axis, and packet arrival time (in 10's of seconds) is on the x-axis.

Figures 2-4, 2-5, and 2-6 show the processing delay, modem queueing delay, and transmission delay respectively for packets on the line between ISI22 and ISI52. The most interesting thing about the plots is the extreme, indeed extraordinary, variability of the packet delays. This variability occurs in both the processing delay and the modem queueing delay. In fact, the extreme variability of the delays is the single most consistent property of the data we have gathered--it is present in all samples. This variability is not what one would expect a priori. Rather, one might expect consistently high delay during periods of high load, and consistently low delay under periods of low load. So it is worthwhile to inquire into the reasons for the variability.

One can never totally rule out the possibility that some result is an artifact either of the data gathering or data analysis techniques. One hypothesis we considered is the following. "Packets may be high priority or low priority. If high priority packets have consistently low delay, and if low

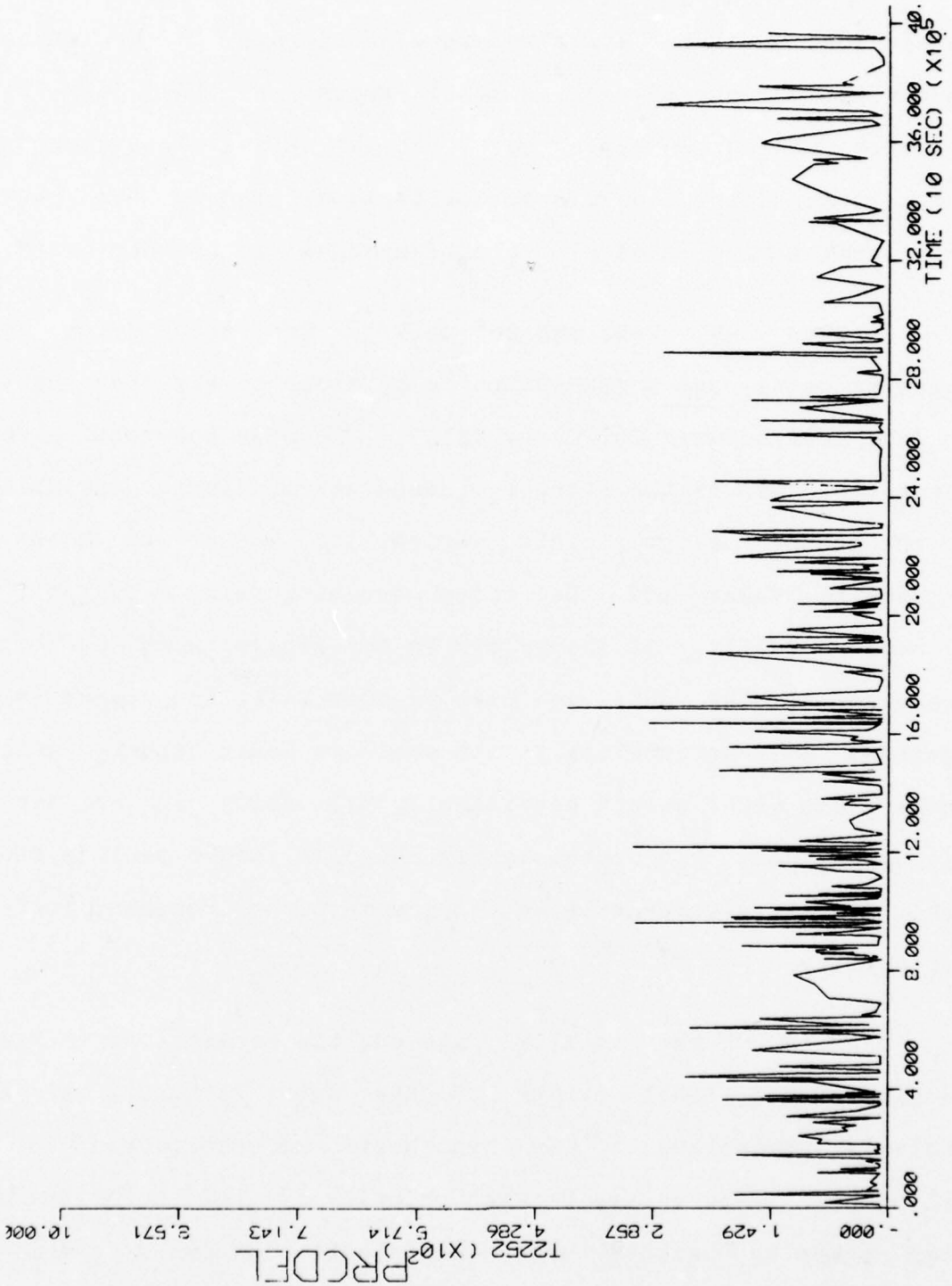


Figure 2-4 Processing Delay, Normal Conditions

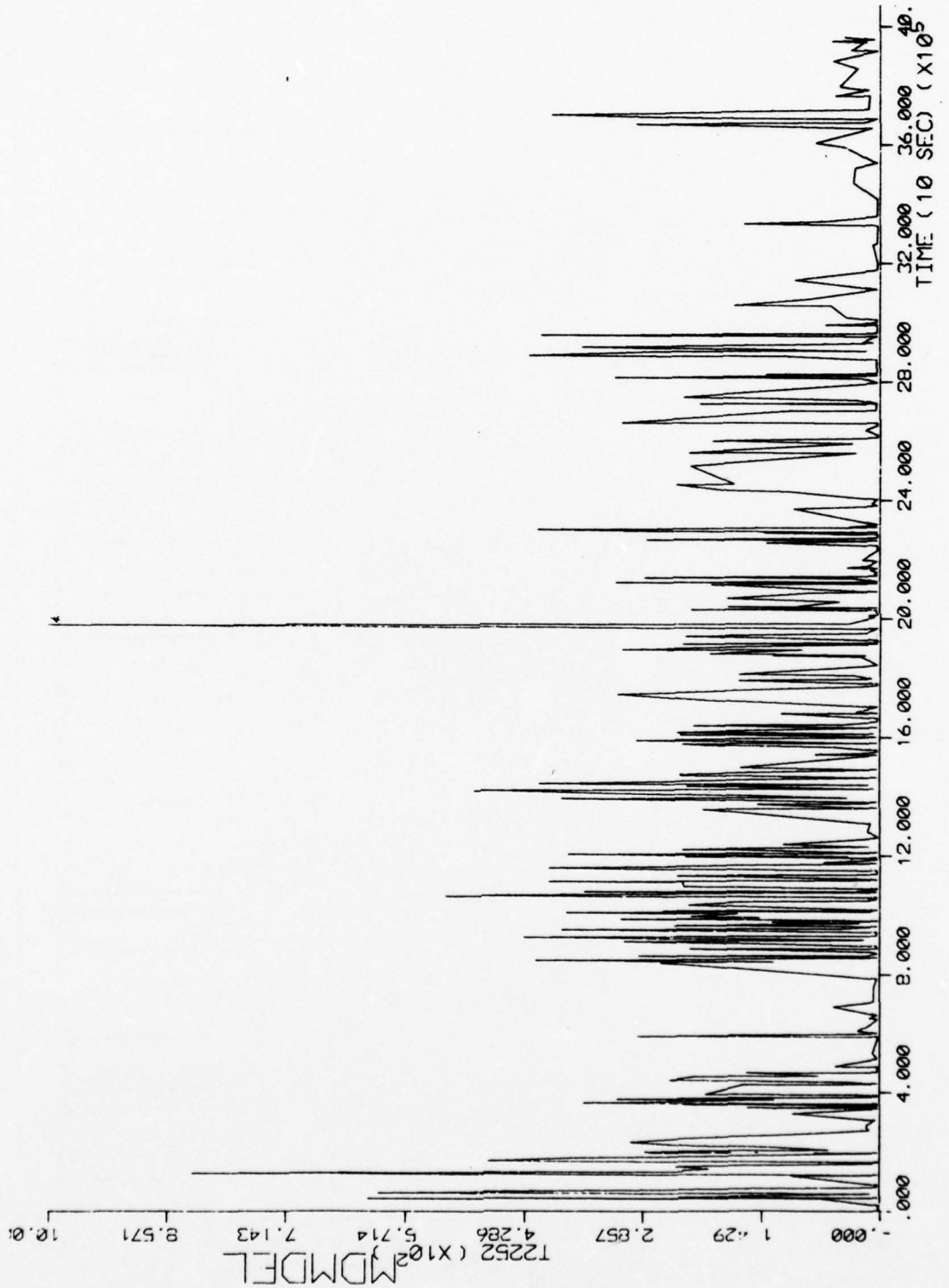


Figure 2-5 Modem Queueing Delay, Normal Conditions

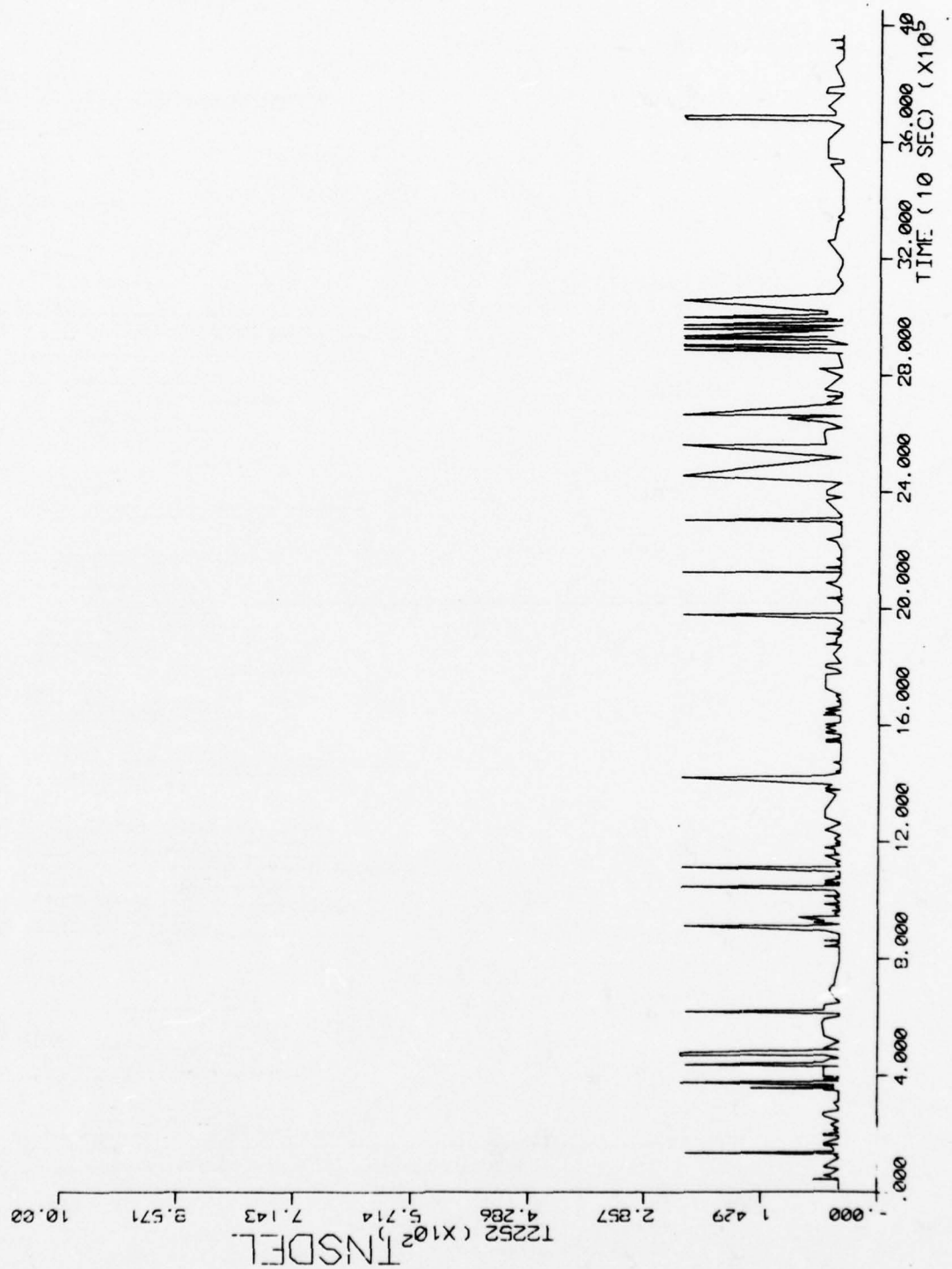


Figure 2-6 Transmission Delay

priority packets have consistently high delay, and if both kinds of packets are freely intermixed in the sample, then of course the delay appears to be very variable. But that variability is not real; it is a result of improper analysis." Of course, this hypothesis could not explain the variability in the processing delay, but only in the modem queueing delay. In order to test this hypothesis we simply plotted the delay for high priority packets separately from the delay for low priority packets. (See Figures 2-7 and 2-8.) The extreme variability is present in both plots, and hence the hypothesis is false.

There are several possible explanations for the variability. The current way of doing routing may contribute variability to both the processing delay and the modem queueing delay. The processor may be interrupted as often as 32 times per second in order to perform the rather lengthy routing computation (15-20 ms.). This can cause the processing delay for packets to vary. Similarly, output on any given line may be interrupted as often as 8 times per second in order to send a routing message, which takes about 23 ms. This can cause the modem queueing delay for packets to vary.

It is also possible that the variability is a natural characteristic of the traffic pattern. Perhaps traffic enters the IMP in bursts, so that queues fill up and then empty out

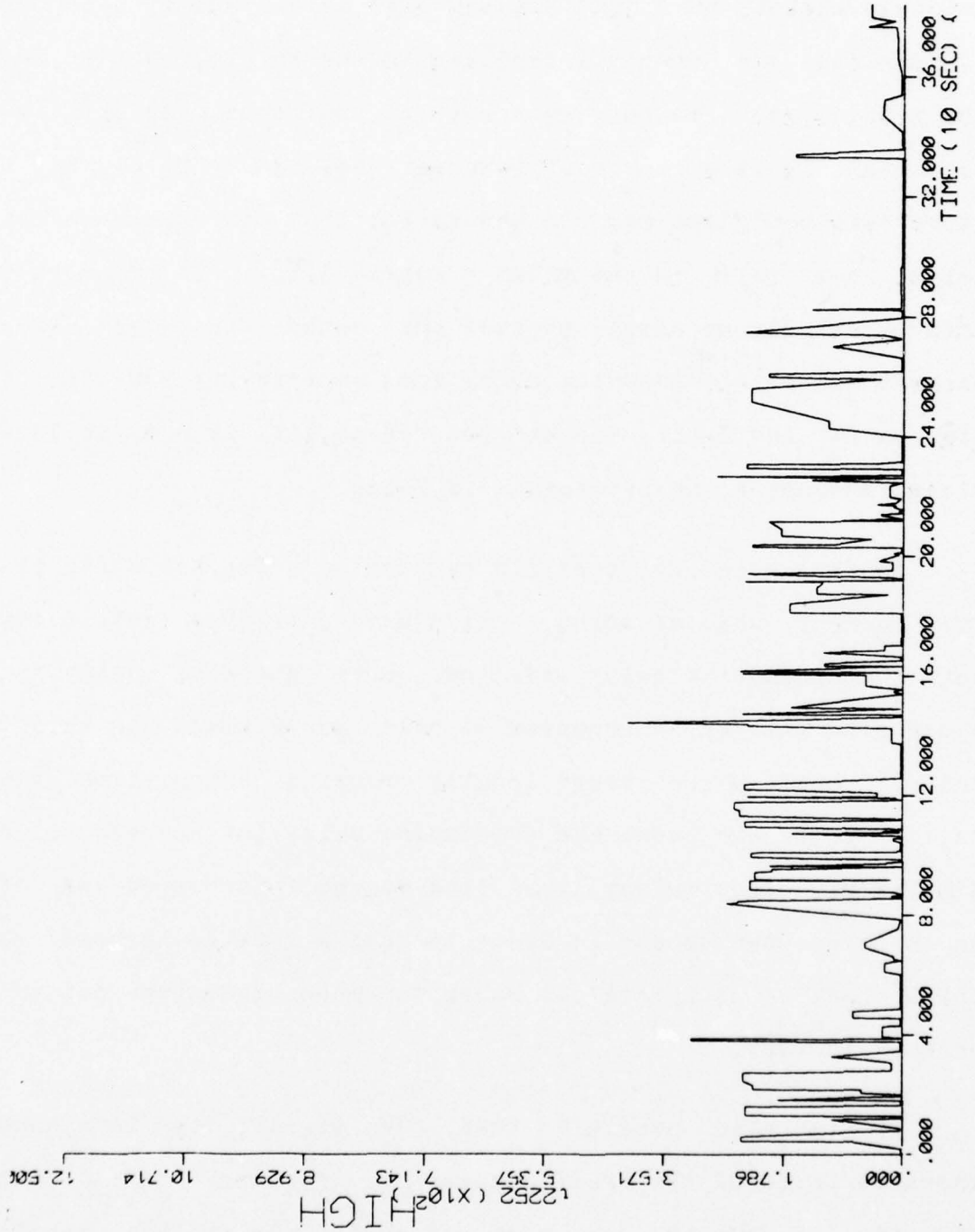


Figure 2-7 Modem Queueing Delay for High Priority Packets (Normal)

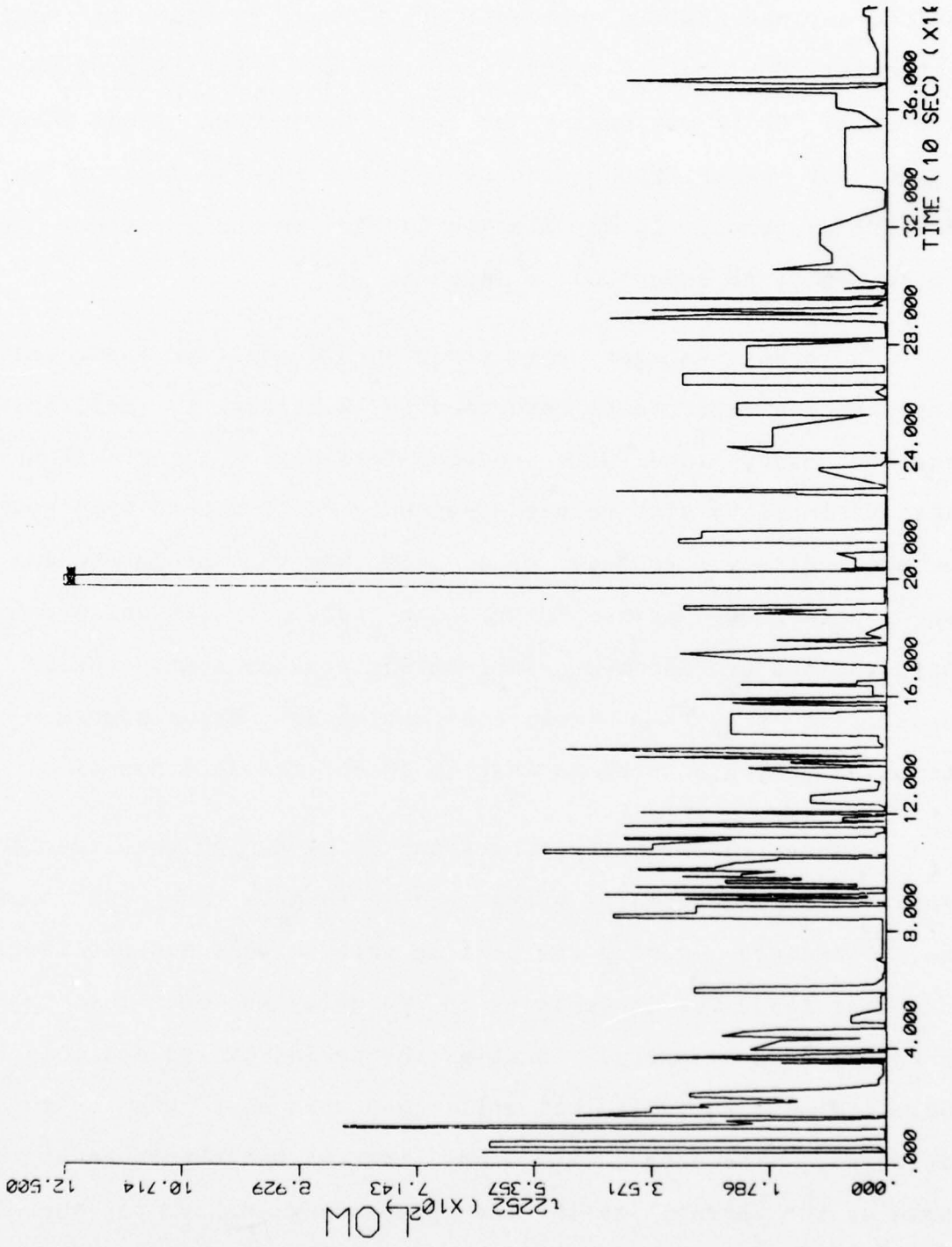


Figure 2-8 Modem Queueing Delay for Low Priority Packets (Normal)

before any new traffic arrives. Or it may be due to various latencies due to the relative timings and priorities of the IMP routines. Or it may have to do with unforeseen (and unknown) throughput restrictions imposed by the End-End protocol on the IMP-IMP protocol. It may also be due to measuring every tenth packet only, as described in Appendix 1.

In order to get some grasp on all this, we conducted the third of the experiments described in Appendix 1, artificially created heavy load with reduced frequency routing. If we see less variability with reduced frequency routing than with routing at the ordinary frequency, we may conclude that at least some of the variability is due to the high frequency with which routing computations are performed and routing updates sent. While the data from this run implicate routing as a major source of the trouble, they also suggest that it is not the sole source.

Figure 2-9 shows the processing delay during the experiment. The artificially created heavy load is roughly from 250 seconds to 550 seconds. During the periods when we were not artificially creating load, the variability in the delay was much less than we have seen previously. During the period of induced load, the delay is more variable, but still less than what was originally observed. Therefore, it seems that we can attribute at least some of the variability in the processing delay to the high

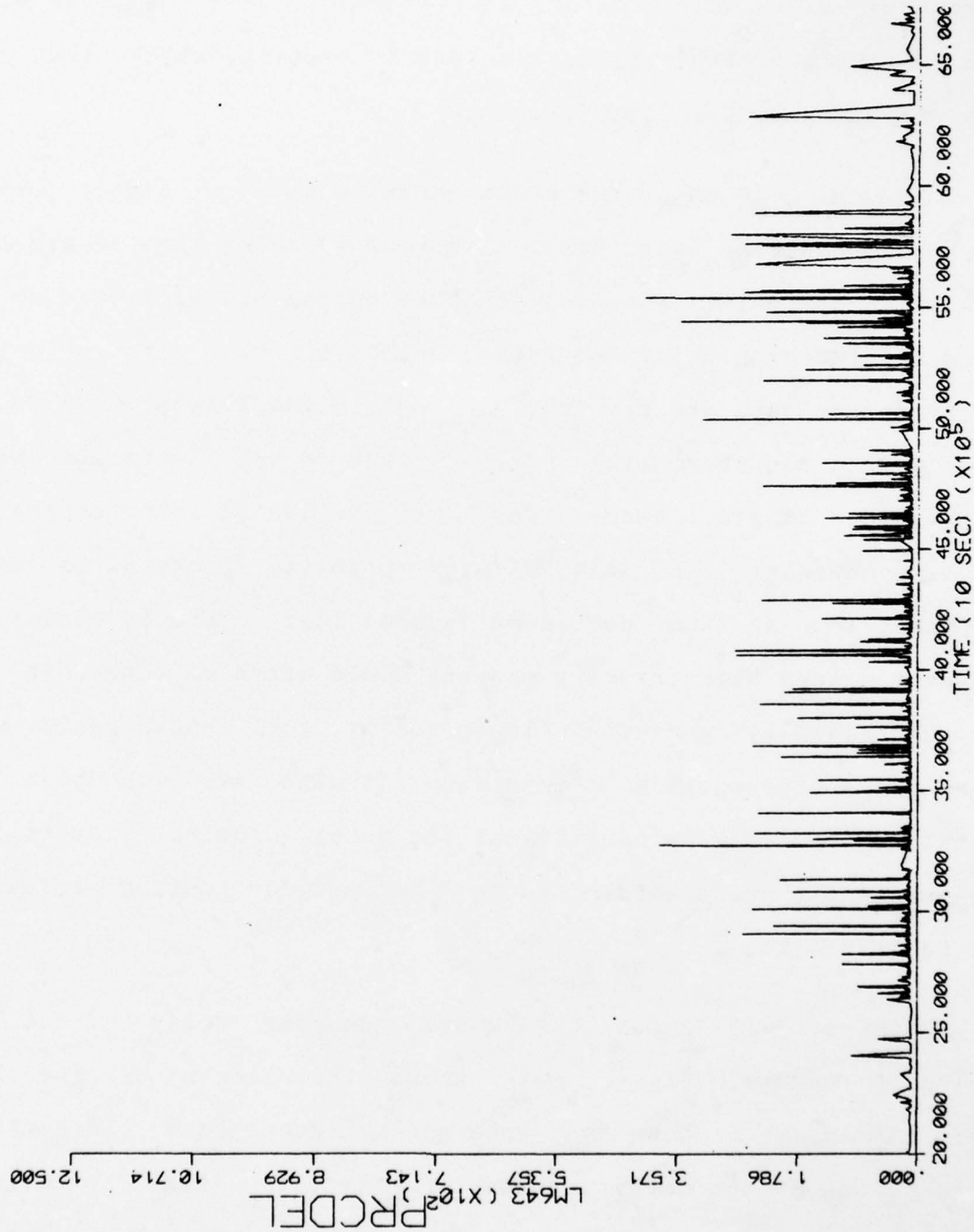


Figure 2-9 Processing Delay (Experiment)

frequency with which routing is performed. It is possible that the remaining variability is also due to routing, which even at its minimum rate is quite frequent.

Figure 2-10 shows the modem queueing delay of high priority packets during the experiment. Figure 2-11 shows the length of the high priority queue. Note that during the periods when we were not inducing heavy load, the delay of the high priority packets was consistently low, with only a few spikes, much fewer than with ordinary routing. These spikes do not correlate with the queue length, hence they must be due to interference by routing messages. The delay of high priority packets is more variable during the period of induced load. This is expected, however, since high priority packets would often have to wait if an artificially generated low-priority full length packet was already in transmission. These data furnish evidence that at least some of the variability of the modem queueing delay is due to the fact that packets often have to wait for routing messages to be transmitted.

Figure 2-12 shows the modem queueing delay of the low priority packets. Figure 2-13 shows the length of the low priority queue. Note that even under induced load, the extreme variability of the delay remains. Further, the delay is highly correlated with variations in the queue length. Furthermore,

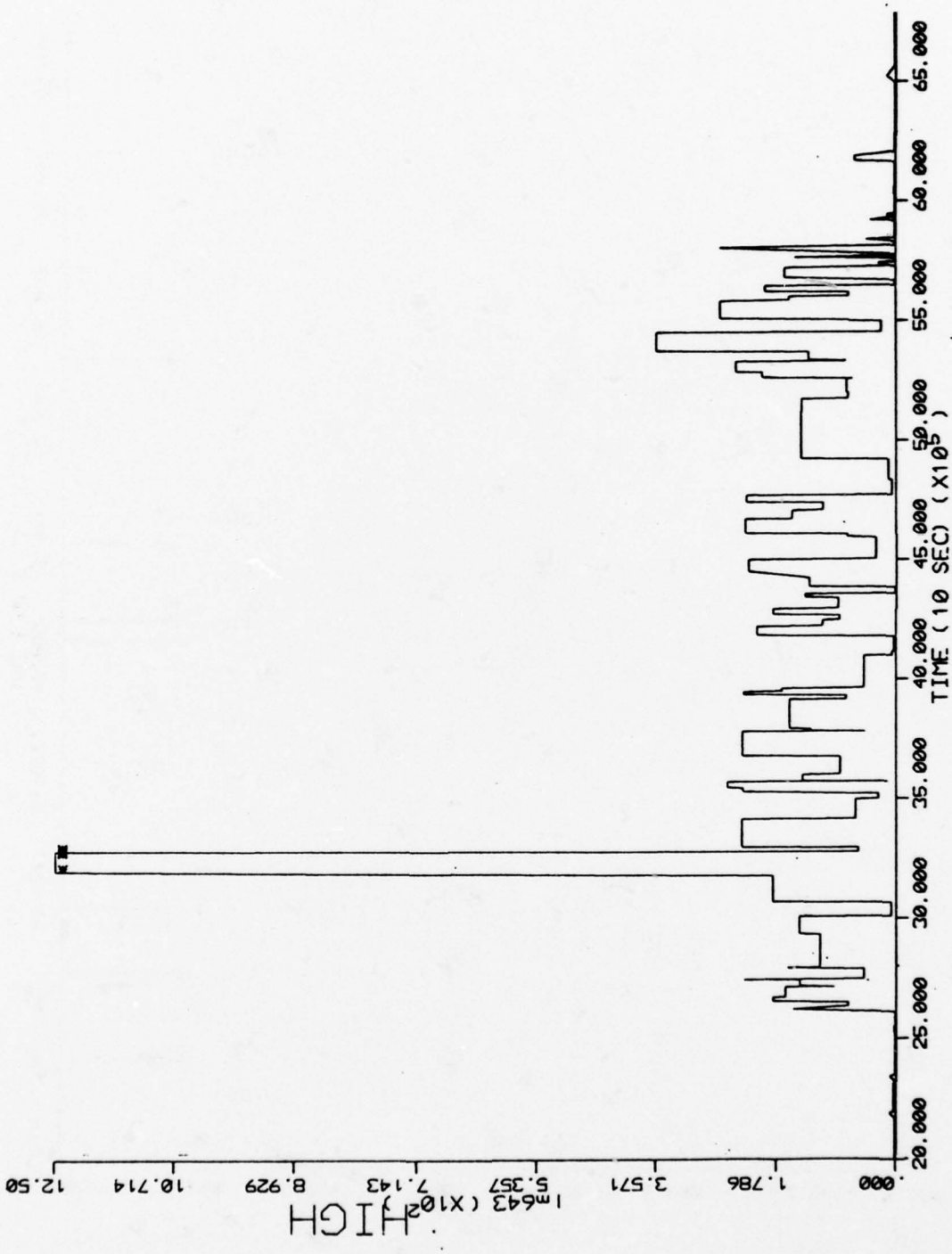


Figure 2-10 Modem Queueing Delay (Experiment)

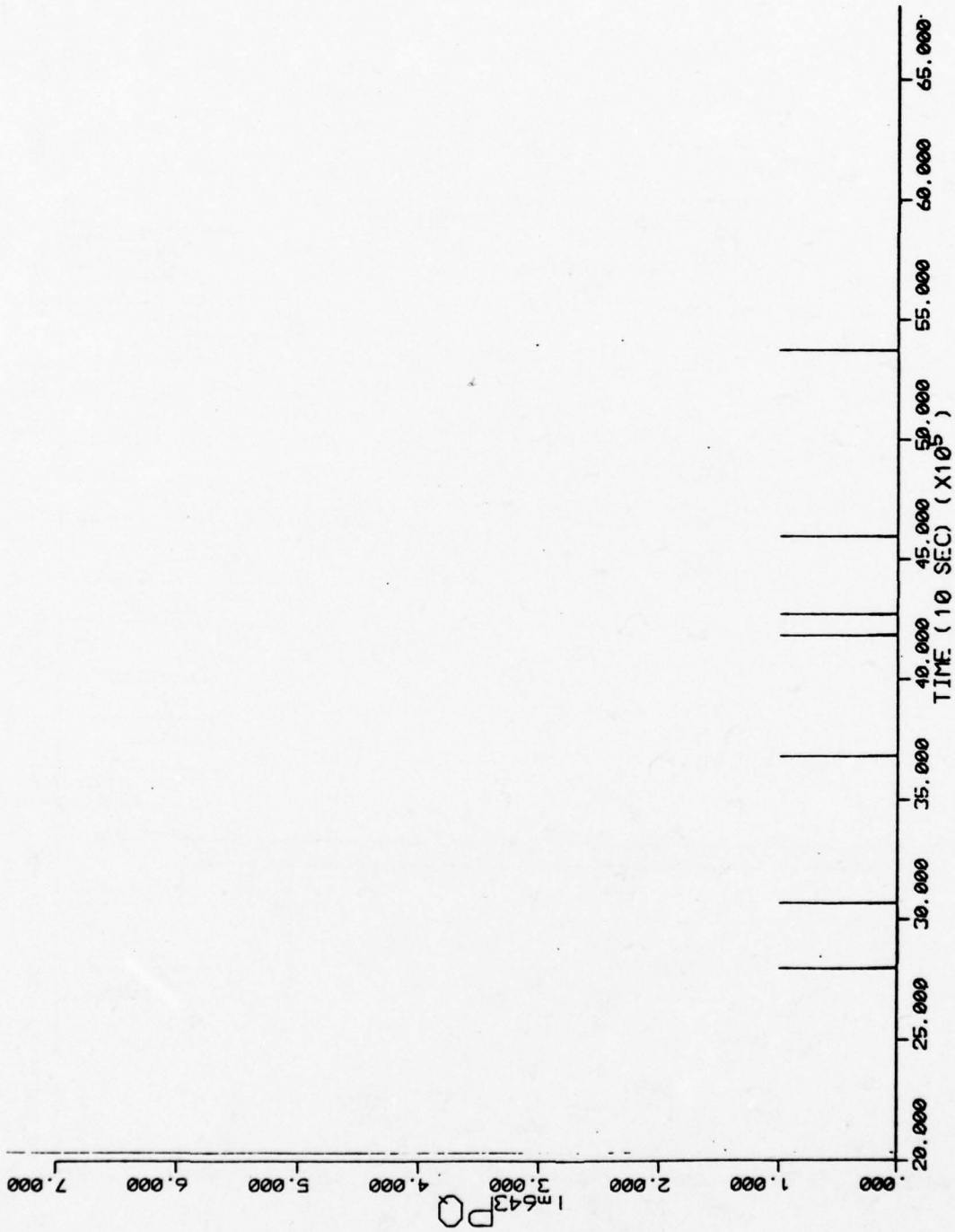


Figure 2-11 High Priority Queue Length (Experiment)

even though we have induced an extremely high load, the queue length falls periodically to zero. We do not yet understand this effect. No doubt the large amounts of processor bandwidth and transmission time required even by reduced routing make some contribution, but it seems clear that other factors are involved also. Discovering the cause may require considerable investigation.

Another interesting point revealed by Figures 2-12 and 2-13 is that instantaneous samples of the queue length show little correlation with the actual delay. Since the current ARPANET routing uses such an estimation technique, it is not systematically adaptive with respect to changes in delay. Its delay measurement facility is inadequate to the task of finding the least-delay path. In fact, no instantaneous measure of delay can be adequate for the purpose of predicting future delay, if the delay is as variable as we have seen. Therefore, a good routing algorithm will have to include an algorithm for smoothing the delay data over time. This is discussed in more detail in Section 4.

Perhaps the main thing to be learned from this data is the important point that the routing algorithm itself may affect the delays through each IMP in hard-to-foresee ways. As has been pointed out, any routing algorithm must include a delay

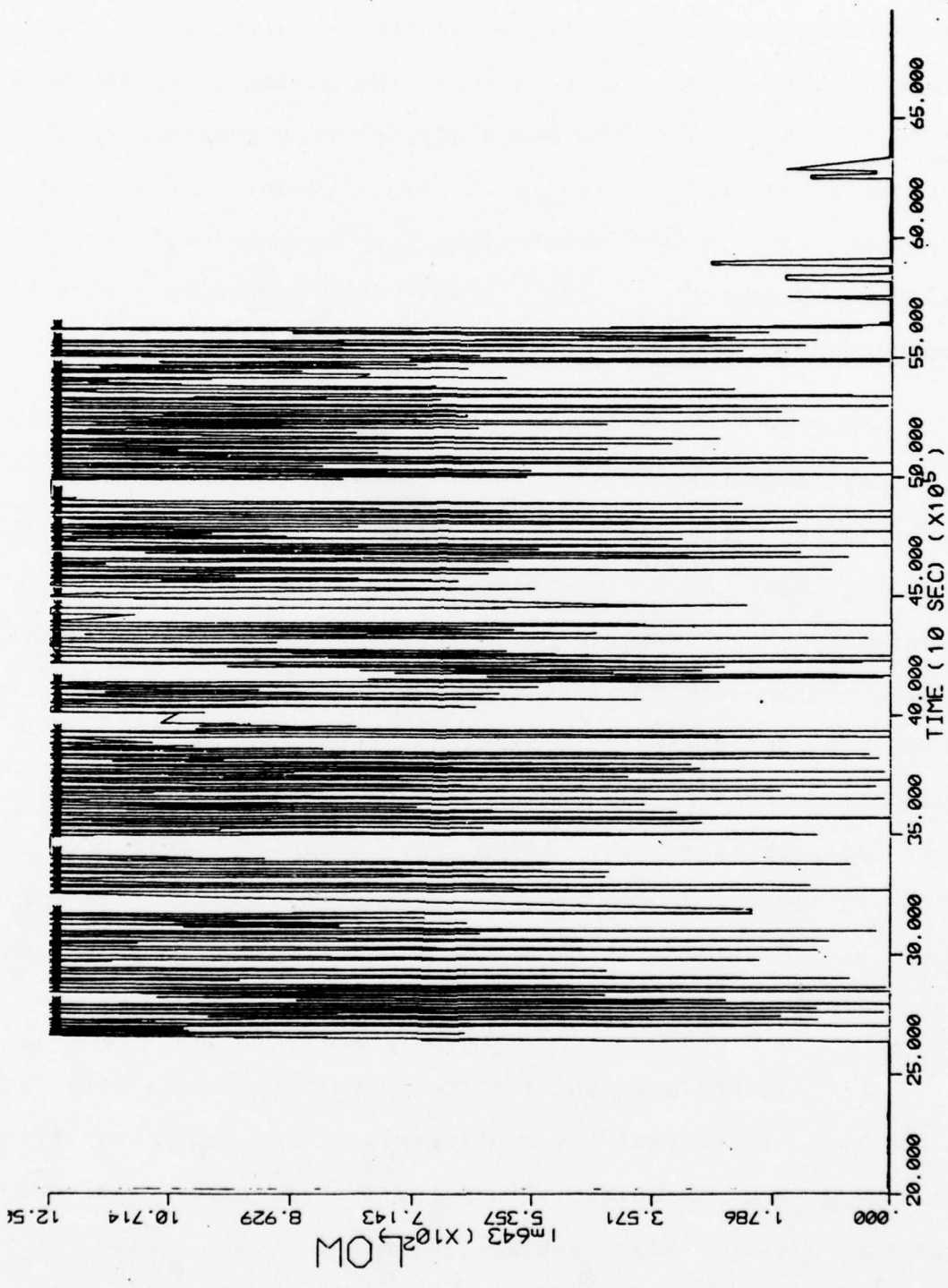


Figure 2-12 Modem Queueing Delay, Low Priority Packets (Experiment)

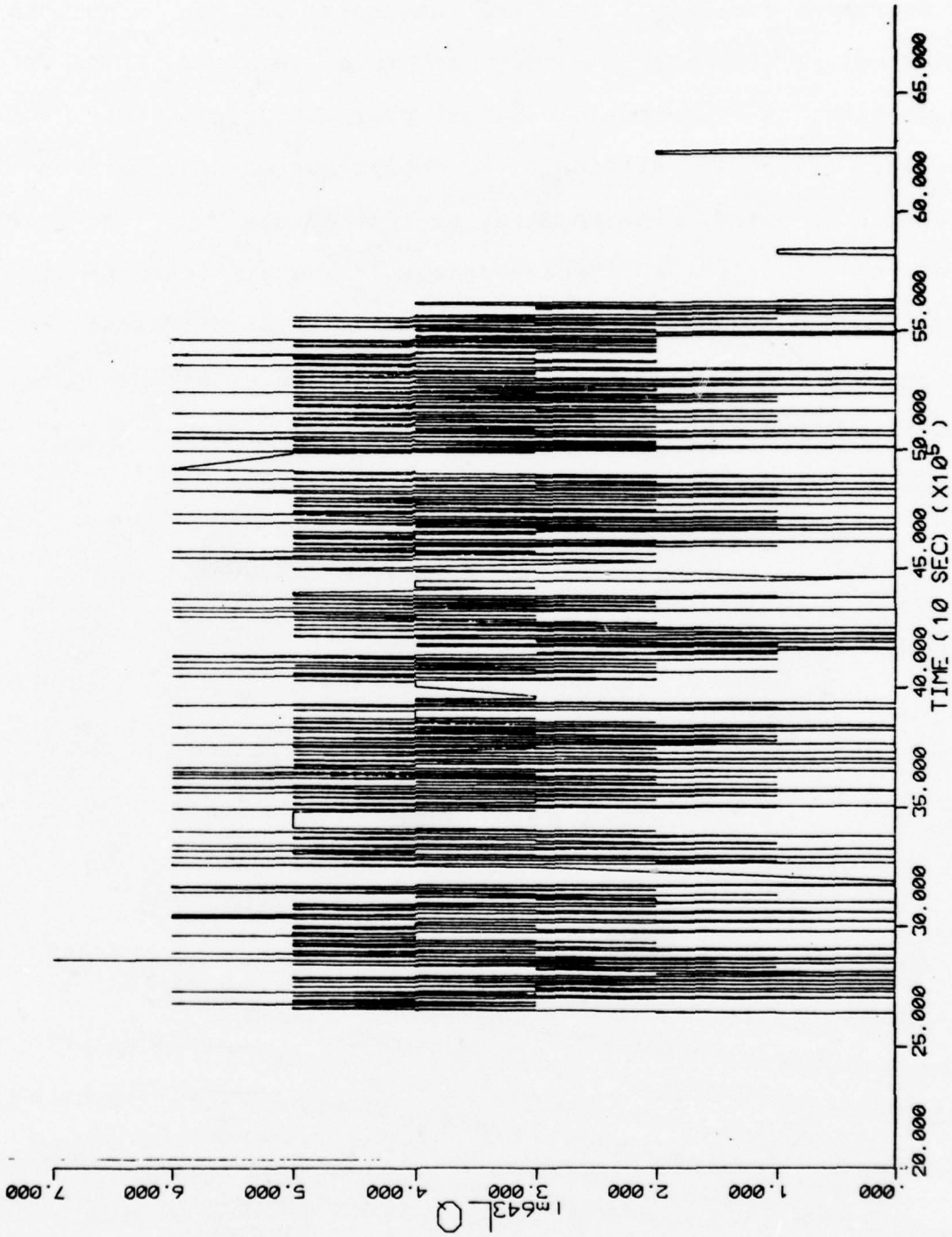


Figure 2-13 Total Modem Queue Length (Experiment)

measurement facility, and the accuracy of the algorithm is constrained by the accuracy of this facility. The current algorithm, by contributing a great deal of variability to the delay, makes it difficult to obtain any meaningful predictive measure of delay. Furthermore, because of the way the current routing algorithm influences delays, it is difficult to use data gathered at present to predict the behavior of different routing algorithms. This means that the task of gathering delay data must continue after a new algorithm is implemented.

2.5 Snapshot Measurement Package

2.5.1 Introduction

The snapshot measurement package, from now on called the SNAP package or simply the package, is a set of routines that can be loaded into an IMP to measure and report key variables and events. It was designed to obtain data about the performance of routing during periods of stress (congestion) and is expected to be used in the analysis of new routing algorithms during the next few years.

It is always difficult to obtain measurements in a network, especially when congestion is present. For one thing, in order to understand the cause of the congestion, we must gather information about what was happening in the network immediately prior to the formation of the congestion. Since we do not know in advance when congestion will arise, some form of continual measurement is necessary. Second, by its very nature, congestion makes it undesirable to transmit measurement data at the time of the event being recorded. The network is clogged and adding new packets filled with measurement data would merely increase the congestion and contaminate the results. Thus it is necessary for the package to remember its data, waiting patiently until the congestion subsides and the network is back to normal before sending the data to a central collection point for analysis.

The mechanism that was developed to meet these requirements of continual generation and storage of data employs a ring of measurement buffers. Routines were written to record the values of key variables and queues in the IMP. The main routine runs at a fixed interval, capturing the state of the machine, and recording notable events that occurred during the previous interval. Each time it runs, it fills the next buffer in the ring with the data it has gathered. The buffer full of measurement data is known as a snapshot. When the package reaches the beginning of the ring, it overwrites the data already in that buffer, data which by now is presumably too old to be of interest. Thus, if the package has N recording buffers, it can remember the state of the IMP for the last N intervals.

When the network is under stress, certain symptoms become evident to the IMP: lines may go down, packets may be discarded or rerouted. The occurrence of one of these events can be used to halt the recording process, thereby freezing the last snapshots taken. The package will stay in this state until the data is collected and the package is reset. Events that freeze the recording process are called triggering events or simply triggers.

Sometimes several important events may occur in rapid succession. It is desirable to have data about each of these

events, yet it may not be possible to reset the package between them. Therefore, the snapshot mechanism has multiple sets of data buffers, each one capable of recording information about one trigger. When the trigger fires, the snapshots in the present set of buffers are frozen and recording is started in a new set of buffers. This process continues until all sets of buffers are used. Since each set of buffers records the events leading up to one trigger, the number of sets is usually referred to as the number of triggers. Thus a particular implementation of the package might have three triggers of two snapshots each--the snapshot immediately preceding the triggering event and the snapshot immediately following it. Our experience has been that the right combination of triggers and snapshots depends upon the particular phenomenon being studied.

2.5.2 Measurement Routines

There are four components of the measurement package: the SNAP loop, the marking routines, the triggers and a patch to the IMP program. The main one, the SNAP loop, is called by the IMP's slow timeout routine and fills a snapshot buffer with data that it computes itself or with data which has been recorded by the marking routines. It also manages the snapshot buffers, recognizing triggers and freezing a set of snapshots. The marking routines are small pieces of code that set flags or

increment counters to indicate the occurrence of a particular event during the previous interval. Each piece is hooked into the IMP program at the point where an action that is to be recorded is initiated or discovered. The trigger routines are much like the marking routines in that they are short pieces of code that are hooked into the IMP program at points where events are recognized. However, rather than merely marking the occurrence of an event, a trigger routine sets the trigger flag so the current set of snapshots are frozen after the next SNAP run.

The SNAP routine is called at the end of every slow timeout run (i.e., once every 640 ms.). It first checks to see if it has any measurement buffers remaining, and if not, it returns immediately. Next, SNAP checks to see whether it is time to record another snapshot. Currently, SNAP only records data every fifth time it is called (i.e., once every 3.2 seconds). If it is not yet time for the next sample, control is returned to the IMP. Otherwise, the data gathering process begins. Various time measures are computed first. Next, an attempt is made to account for all the IMP's buffers. Following this, SNAP scans each line and records various data about its state.

The final action of the SNAP code is to check whether any triggering events happened during the last interval. If not,

SNAP returns control to the IMP. If a triggering event has occurred, the data is frozen so that it can be collected. With the new data properly recorded and frozen, the SNAP routine returns to the IMP.

The marking routines record asynchronous events for the SNAP routine, thereby enabling it to see whether some event has occurred since it last ran. (Note that the events marked here are not necessarily triggering events.) The trigger routines monitor events which are indicative of some sort of disturbance or malfunction; the purpose of the trigger is to freeze the latest set of snapshots. The occurrence of a triggering event causes a message to appear on the NCC log, thereby alerting the network controllers to collect the snapshots. Many network problems, especially those which involve routing, involve interactions between IMPs. Therefore, the package has a mechanism to induce triggers in two IMPs at approximately the same time by sending a special packet to the adjacent IMP.

2.5.3 Conclusions

The SNAP package has already proved its usefulness, and we expect to continue the evolution of this flexible, real-time data gathering mechanism. One clear conclusion is that a modular approach has been very helpful here, since the events being

monitored and the data being gathered are subject to the continual change as analysis and development proceed.

3. LINE UP/DOWN PROTOCOL

3.1 Introduction

As noted in Section 2, we have observed a number of line mismatches, some of them lasting for considerable lengths of time. Therefore, some analysis of the present line-up counter was carried out. This analysis indicated some rather severe weaknesses, and in the course of considering alternatives, it became clear that the line-down counter could also be improved. This section first outlines present line up/down counters and the difficulties inherent in these procedures. Then some general goals are described and a new solution is proposed that will meet the stated goals. This solution involves the use of new counters and the addition of a new state to the line up/down protocol.

3.2 Existing Counters

Currently, lines exchange Hellos and I Heard You's (IHYs). The former are long packets (approximately 1100 bits), whereas the latter are short (approximately 150 bits). Depending upon line utilization, an IMP sends one to five Hellos per tick (640 msec). The IMP will get an IHY if (1) at least one Hello gets

through without error, and, (2) the IHY is received without error. A line is brought down if an IMP misses IHYs in five consecutive ticks. After some dead time (10 ticks) to insure that the other side of the line is brought down, an up/down counter is used to determine when to bring the line up. The up/down counter is initialized to C = -60 and then follows the algorithm:

```
if (-60≤C<0) then
   if (get IHY during interval) then (C<- C+1)else(C<- C-1);

if (-240<C< -60)then
   if (get IHY during interval) then (C<- -60)else(C<- C-1);
```

Finally, if C reaches -240, the line hardware is reset and C is reset to -60.

There are several drawbacks to the present scheme:

1. Poor lines are brought down too slowly. If the probability p of missing an IHY is 0.2, then the expected time to bring the line down (i.e., to miss five successive IHYs) is approximately 40 minutes. In fact, even a very poor line with p = 0.4, will remain up for an average of 1.7 minutes.

2. Poor lines are brought up too quickly. Clearly, if the probability of missing an IHY is less than 0.5, then the

up/down counter will tend to count up toward zero. For example, if p = 0.2, then the expected increment to the counter is 0.6 during each tick, and the line will, on the average, come up in about 100 ticks (1 minute).

3. Line mismatches can occur. The different lengths of the Hello and IHY lead to an asymmetry that can cause line mismatches if communications in one direction on a line are not as reliable as communications in the opposite direction. Consider the following situation: Line AB and its associated hardware carries traffic from node A to node B; line BA carries traffic in the reverse direction. Assume that line AB is perfect (no errors occur), but that line BA introduces errors such that Hellos are received in error with probability 0.5, but IHYs, about 1/6 the size, are damaged with probability 0.1. This would correspond for example to random bit errors occurring with probability 0.0006. Then, shortly after both lines are brought down, line AB will be brought up again, but BA will remain down for a very long period. Analogous situations can result in shorter, but still undesirable mismatches.

Figure 3-1 is a graphic illustration of the last point. The horizontal and vertical axes correspond to random bit errors, but the same principle applies if these axes were labeled in terms of packet errors. The different regions in the figure correspond to different durations of line mismatches. For example, if both lines have error probabilities less than 0.0001, then both sides will come up within about 10 ticks of each other. On the other hand, if the error probability of each line is approximately .0003, then both lines will tend to come up, but they will be mismatched for 50 or more ticks.

3.3 Goals

An ideal set of line up/down procedures would instantly bring a line down when it becomes bad and would immediately bring a line up again when it is restored to good health. Obviously, such ideal goals cannot be met in practice. The table below summarizes the ideal goals and the sort of goals that we can realistically expect to achieve. Ideally we would like to specify very sharp transitions between the up and down states. The particular "realistic" entries in the table can, of course, be modified, but the idea is (1) to bring poor lines down quickly and keep them down longer than the period over which a line is usually bad; and (2) to rarely bring a good line down but to bring a good line up with reasonable speed.

L i n e Q u a l i t y

	Nominal	Poor
Bringing Line Down	ideal: never realistic: less than once/week	ideal: instantly realistic: within about 10 sec.
Bringing Line Up	ideal: instantly realistic: within about 1 min.	ideal: never realistic: less than once/hour

The problem now is to quantify the terms "nominal line" and "poor line". It is best to divorce our criteria from line characteristics and instead to relate them to desired network performance characteristics, via the packet error rate. For example, as the packet error rate increases, delay increases and throughput decreases. In addition, an increasing fraction of control messages are also lost, further degrading performance. Since the packets in the network are of different sizes, it is difficult to specify a single packet error rate that characterizes a poor line. A reasonable criterion is therefore to define a nominal and a poor line in terms of the errors experienced by the packets used in the line up/down protocol itself.

With event-driven updating, routing messages alone cannot be used in the up/down protocol. Instead, a special purpose packet

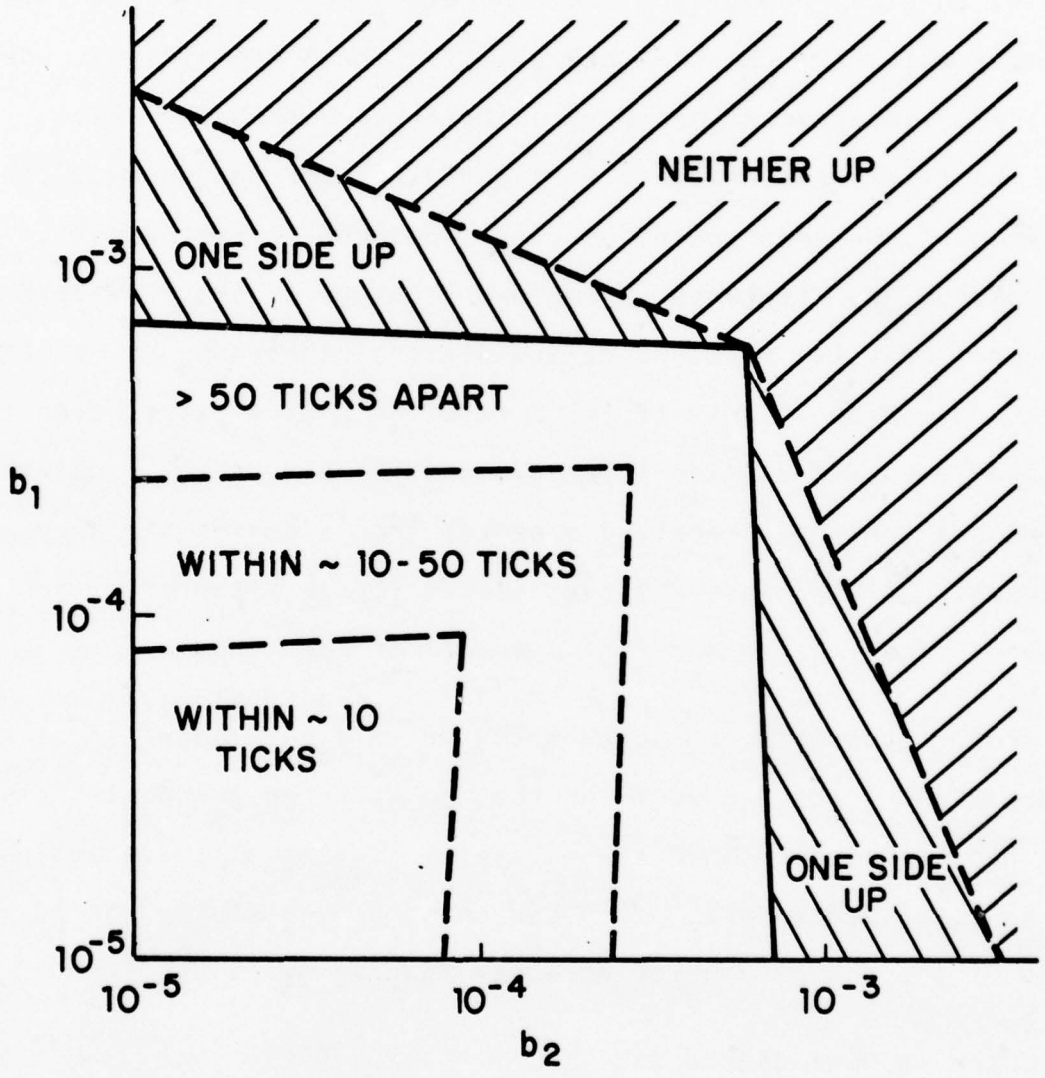


Figure 3-1 Line Mismatch Regions

must be exchanged periodically to allow nodes to determine line status, and these packets should be small in order to reduce the impacts on line bandwidth and queueing delays. Since these packets will be considerably smaller than the maximum packet size, and since packet error rate increases with packet size, we must be fairly conservative in specifying poor and nominal lines in terms of the error rates of these packets. Furthermore, we may choose to implement a two-way strategy on the line up/down packets; that is, we might define correct reception of a line packet at node A only if (1) A receives a line packet from its neighbor B during a given interval, and (2) B's packet indicates that B had correctly received a packet from A during the previous interval. A two-way strategy allows closer synchronization of two ends of a line at a slight increase in complexity.

For illustrative purposes below we will occasionally use the values $p = 0.1$ and $p = 0.001$ as the packet error probabilities of poor and of nominal lines respectively. The specific values used in the final design will of course be chosen carefully and on the basis of the entire routing strategy that we develop.

3.4 Line Up/Down Counters

The type of counter we shall consider will be called a "k out of n counter" and, for convenience, will be denoted by (k,n).

This "counter" is said to trigger if within any block of n or fewer intervals, k events occur. For a line going down, an event would be missing a packet, and thus the line would be brought down if k packets were missed in n or fewer intervals. For a line coming up, the appropriate event is receiving k packets in n intervals. The (k,n) counter is a fairly general type of counter and is one that can be implemented in a straightforward manner.

Another possibility would be to use a counter that counted +1 if a certain event occurred and -A if the event did not occur. Such a generalized up/down counter would trigger if some threshold T were crossed. This counter is also easy to implement and it is roughly equivalent to the (k,n) counter in the sense that we can choose values of A and T that give performance similar to that of a particular (k,n) counter. A minor drawback of this generalized up/down counter is that it has a longer memory than the (k,n) counter. That is, the up/down counter can tend to creep in one direction and therefore might not truly reflect the most recent history of the line. On the other hand, the (k,n) counter has a memory of exactly n intervals. For this reason we shall consider only the (k,n) counter.

The performance parameter of particular interest is the expected number of intervals until the counter triggers. With a little thought, it becomes apparent that this type of counter can

be expressed in the form of some standard queueing theory problems, namely,

-- A single server queue with Poisson arrivals and constant server time equal to n units; what is the expected time until the queue reaches a length of k?

-- k servers with Poisson arrivals and constant server time n; what is the probability that the next arrival will be blocked (i.e., cannot be served immediately)?

Although these queueing theory problems can be stated simply, it turns out that they are examples of some classic unsolved problems. The difficulty lies in the constant server time. Thus, we cannot expect to obtain a general solution to the (k,n) counter. Nevertheless, we have obtained a few useful analytic results for some specific counters of interest. If additional results were necessary, simple simulations were conducted.

3.4.1 Line Down Counter

Obviously, we cannot base the line down procedure on missing a single packet since such an event occurs often enough that a line would then be brought down too frequently. We shall therefore consider the next simplest counter, a (2,n) counter, and determine whether such a counter is adequate for bringing a line down.

Define

p = Pr [line packet missed during interval]
$q = 1-p$

E(N) = expected number of intervals for event (line UP or line DOWN)

Then, it can be shown that for a (2,n) counter

$$E_2(N) = \frac{2-q^{n-1}}{p(1-q^{n-1})} \qquad (3\text{-}1)$$

For small p, the above expression reduces to

$$E_2(N) \sim P^{-2}\frac{1}{(n-1)} \qquad P<<1 \qquad (3\text{-}2)$$

This relation can be derived intuitively, since the probability that a second packet is missed in one of the (n-1) intervals following a missed packet, is approximately p(n-1). Hence, the expected interval until the occurrence of the event "miss 2-out-of-n" is (1/p)*(1/p(n-1)). This equation shows that even if n can be chosen as small as 10, and if p = 0.0003 (optimistic for a nominal line), then $E_2(N)<10^6$, or about one week if each interval is one tick (640 msec). In other words, a (2,n) counter will bring good lines down too often.

We shall therefore consider a (3,n) counter. First, some simple calculations are given to justify the suitability of this counter. An intuitive argument similar to the one used above, yields the expected number of intervals for bringing a line down if p<<1:

$$E_3(N) \sim \frac{2}{p^3(n-1)(n-2)} \qquad (3\text{-}3)$$

If $p<10^{-3}$, then $n\leq 46$ ensures that $E_3(N)>10^6$. Another quantity of interest is the probability that a line will go down within, say, the first $m\leq n$ intervals after the line becomes bad. Since three packets must be missed, the line will be brought down in exactly k intervals if the k-th packet is missed and if two of the first (k-1) packets are also missed. Hence

$$\Pr[\text{DOWN within m intervals}] = \sum_{k=2}^{m-1} \binom{k}{2} p^3 q^{k-2} \qquad (3\text{-}4)$$

This expression shows, for example, that if p = 0.1 and $n\geq 20$, then with probability 0.32 the line will be brought down within 20 intervals; similarly, if $n\geq 40$, the line will be brought down within 40 intervals with probability 0.79.

More detailed analysis of the (3,n) counter is quite difficult, and therefore simulations were performed in order to obtain specific quantitative data. The solid curves in Figure 3-2 show simulation results for $E_3(N)$ as a function of p for several values of n; the curve for infinite n is based upon the theoretical result, $E_3(N)=3/p$; and the dashed curve represents a(2,5) counter, as given by Eq. (3-1). If the design points are p=0.1 and p=0.001, then the (3,40) counter just meets the

criterion of bringing a good line down on average of only once per week. However, the measured standard deviation of the number of ticks to bring a line down is quite large, approximately equal to the mean. A more conservative approach is therefore to use a smaller value of n in order to reduce the likelihood of an undesirable sequence of events, such as bringing a particular good line down several times in one week. The only penalty in this approach is a slight increase in the time to bring a bad line down. the following table illustrates such a trade-off:

	$E_3(N)$	
	n=40	n=20
p=0.1	32	41
p=0.001	1.5×10^6	6.0×10^6

Thus, for these parameters, a one-third increase in $E_3(N)$ at p=0.1 corresponds to a four-fold increase at p=0.001. Note also that a (3,n) counter responds faster to very bad lines (p=1) than the existing counter.

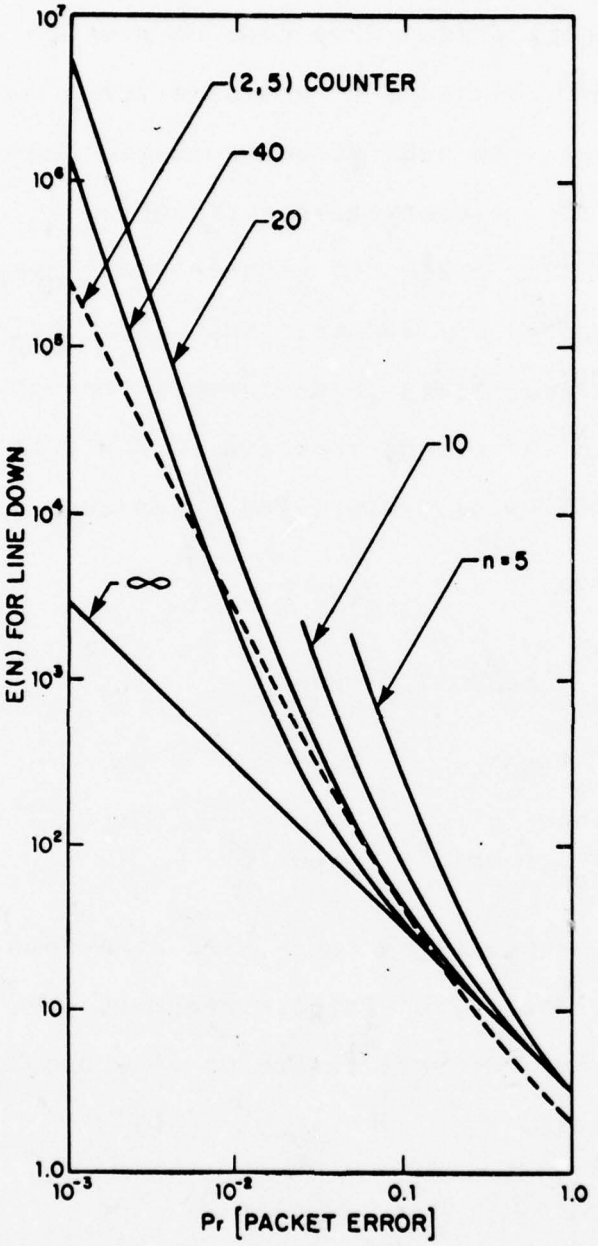


Figure 3-2
Simulation results, shown as solid curves, for (3,n) counters

3.4.2 Line Up Counter

First consider an (n,n) counter (i.e., a consecutive counter.) The expected number of intervals until the occurrence of a success-run of length n can be expressed in closed form, and is given by

$$E_n(N) = \frac{1-q^n}{pq^n} \qquad (3\text{-}5)$$

Here a "success" is receiving a packet correctly (probability q). Equation (3-5) is plotted in Figure 3-3 for $E_n(N)<10^3$. In order to ensure that with p=0.1, E(N)>5600 (once/hour with 640 msec intervals). we must choose $n \geq 61$. If, for example, n=61, then with $p=10^{-3}$, E(N)<63. Thus, there is a reasonable range of n that will satisfy our goals for the line up counter.

It is also possible to use an (n-1,n) counter to meet these goals. However, it is clear from the goal for bringing up poor lines that such a counter must have a larger n than an (n,n) counter. Since the time to bring a good line up will thus be increased, and since there are no concomitant benefits, the (n,n) counter is preferable.

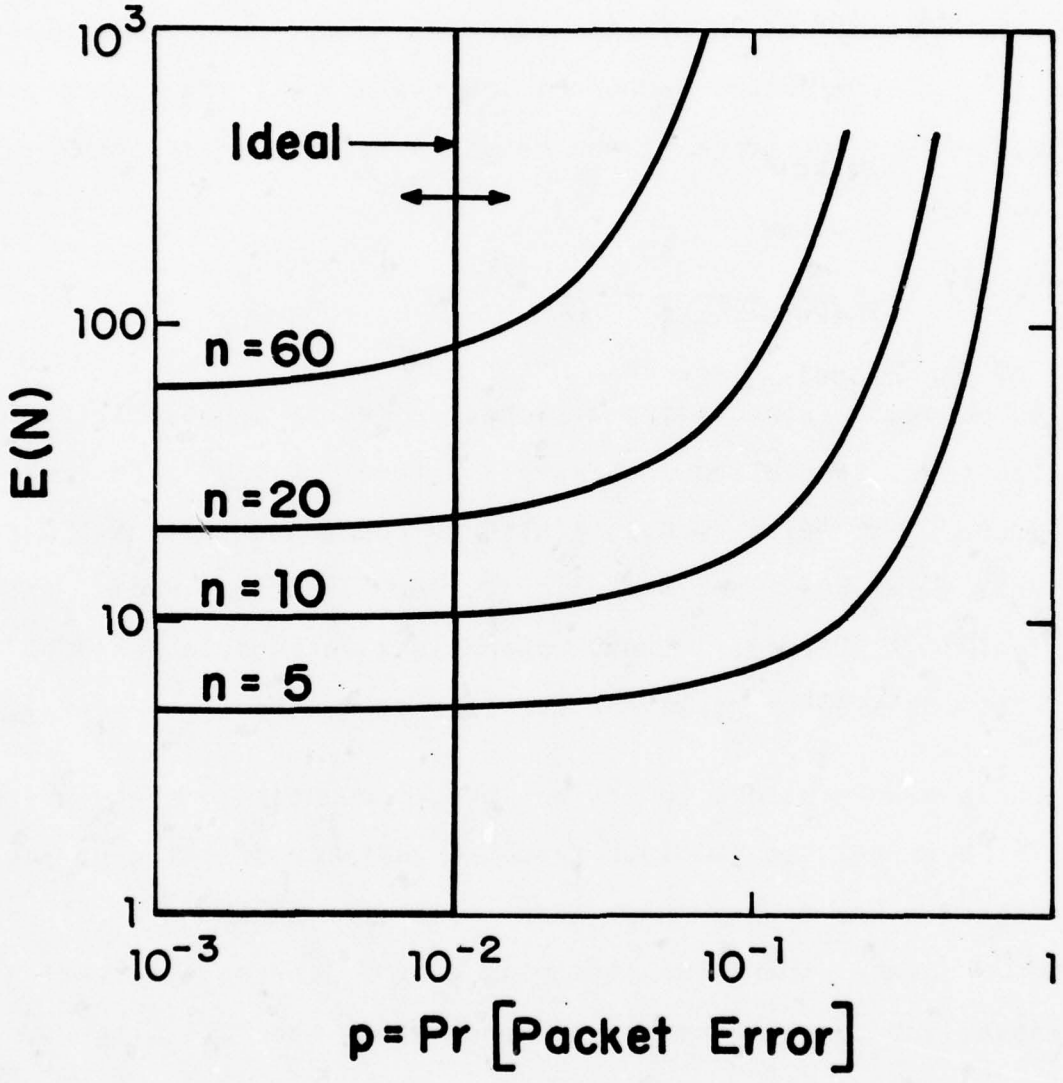


Figure 3-3
Performance of consecutive line up counter

3.5 READY State

Although the use of improved counters can significantly reduce the likelihood of a line mismatch, they cannot easily eliminate mismatches entirely. Therefore, a new state--the READY state--is being added to the line-coming-up protocol. Rather than always being either UP or DOWN, a line will be either UP, DOWN, or READY. The new state can be defined by its transitions:

1. A line makes the transition from DOWN to READY when its line up counter triggers.

2. A line makes the transition from READY to DOWN under the same conditions as it makes the transition from UP to DOWN.

3. A line makes the transition from READY to UP when it receives a packet indicating that the IMP on the other end of the line has declared the line to be non-DOWN (i.e. either READY or UP). Furthermore, when a line makes this transition, a special packet is sent to notify the neighbor that the line has been declared non-DOWN. This ensures that the neighbor will have a chance to bring the line up before receiving any data packets.

Except for the transitions in 2) and 3) above, a line in the READY state is treated as if it were DOWN.

The proposed protocol modification will prevent sub-standard lines from coming up in one direction only. For normal lines, it may lengthen slightly the time it takes for the line to come up. However, whereas it is important to keep poor lines from coming up in one direction, there is no important reason to bring a good line up in the shortest possible time.

3.6 Conclusions

The principal results of this section are that

1. A (3,n) counter should be used to bring a line down.
2. A consecutive counter should be used to bring a line up.
3. A READY State can be used to eliminate line mismatches.

Equations (3-3) - (3-5) and Figures 3-2 - 3-4 can be used in the final design, after the sizes, contents, and desired error rates for the packets used in the line procedures have been established.

In addition to using a line-down counter, it may be desirable for each IMP to measure the error rate of real traffic, bringing the line down if the rate exceeds some threshold. Such a procedure would ensure that poor lines are brought down even if for some reason the line protocol packets tend to be received correctly.

4. DELAY MEASUREMENT

4.1 Better Measures of Network Delay

A crucial issue in the design of any routing algorithm is the design of a facility to provide meaningful measures of delay. Two particular issues are:

1. Making the measurements accurate
2. Making the measurements predictive (i.e., good estimates)

The key question is: how should the IMP estimate delay? There are two possibilities:

1. Measure the actual delay experienced by packets
2. Measure the factors determining this delay (line speed, packet length, line utilization, etc.) and estimate delay

4.1.1 Measuring Delay Directly

The measurement of end-to-end delay is complicated by the fact that two IMPs cannot readily keep time synchronized between them. An alternative, therefore, is for each IMP to measure delay by time stamping packets on reception and, just before

transmission, to calculate how long the packet had remained in to that IMP. The next IMP could add in the speed of light delay and the transmission delay to this delay before repeating the same process. Thus, total transmit delay can be accumulated. An 8-bit field could be added to the packet header to accumulate delay from 0 to 6.5 seconds in units of 25 milliseconds. Speed-of-light delay between IMPs can be measured by:

1. Saving the time at which a null packet with a given identifier is sent out;

2. Waiting until that null is acked, and computing the time interval;

3. Subtracting any other delays, such as transmission delay for the null and its ack, and dividing the remainder by 2;

4. Re-performing steps 1-3 enough times (once a minute? 5 minutes?) to arrive at a minimum value representing the speed of light delay.

Alternatively, we could assume 5 milliseconds for land lines and 275 milliseconds for satellites, and we would be reasonably close.

A more realistic approach to computing total path delay in the context of a routing algorithm for the ARPANET would be for each IMP to compute one-hop delays to each adjacent IMP, and to use those as inputs for the routing update process. The routing computation would take these values and deliver total path delays. An IMP can compute its own delay to send a packet to an adjacent IMP without assistance from the adjacent IMP as follows:

-- Just before transmission, calculate how long the packet has remained in the IMP (by time stamping, as noted above).

-- Add in the speed-of-light delay to the neighbor (as above).

-- Add in the transmission delay to the neighbor, derived by a table lookup based on packet length and line speed.

This approach has the advantages of not requiring additions to the packet header or inter-IMP control traffic.

4.1.2 Estimating Delay Indirectly

Delay from one IMP to another can be calculated from the equation

$$\text{Delay} = (\text{Speed of Light Delay}) + (\text{Processing Delay}) + \frac{(\text{Average Packet Length})}{(\text{Circuit Bandwidth})} \times \frac{(\text{Line Utilization})}{(1 - \text{Line Utilization})}$$

assuming an M/M/1 queue. The first two terms can be estimated directly; the last term requires more careful estimation, and also the possibility of multiplies and divides, which the IMP does not do very well. A simple approximation would be to:

1. Assume we know circuit bandwidth to be one of a small number of possibilities (9.6 Kbs, 50 Kbs, 230.4 Kbs).

2. Assume we know average packet length (it is about 250 bits for the net as a whole over the last several years).

3. Measure line utilization to within 10%.

Then one can construct a table like this:

Utilization	9.6 Kbs	50 Kbs	230.4 Kbs
10%	1		
20%	3	1	
30%	5	1	
40%	9	2	
50%	13	3	1
60%	20	4	1
70%	30	6	1
80%	52	10	2
90%	117	23	5

Delay (milliseconds)

This table reduces all the multiplies and divides to a simple lookup based on line speed and utilization.

The shortcoming of this approach is that it does not permit the IMPs to include the effect of different packet lengths for data flowing over different network lines. Delays could vary from half as big to four times larger, depending on the actual distribution. Furthermore, the assumption of an M/M/1 queue is not completely accurate, so the equation itself introduces inaccuracies.

To summarize, it seems that the indirect calculation of network delays is difficult to do with the required accuracy, especially considering the limitations of the IMP as a number cruncher.

4.2 Smoothing Algorithms

How should the IMP derive a believable estimate of delay from a number of noisy, fluctuating samples? Expressed differently, how can the IMP determine when delay has changed by a significant amount (say 10%) which should be communicated to other IMPs to determine if new routes should be used? There is substantial technical literature on the subject of smoothing algorithms, also known as "filters" in signal-processing jargon. Some well-known smoothers are:

-- a low pass filter, a linear smoother which operates to remove high frequency components; e.g., an average of the last n samples.

-- a high pass filter, a similar smoother which removes the low frequency components (including the mean) - not useful for us.

-- a nonlinear filter, which is capable of preserving sharp discontinuities in the data (unlike the linear smoothers), while still filtering out noise.

Recently, researchers have suggested the use of a running median rather than a running mean. In certain applications, the median of the last n points, say 5 or so, is a very good method

for smoothing out most noise and for following major discontinuities in the delay values (recognizing new delay levels). The paper "Applications of a Nonlinear Smoothing Algorithm to Speech Processing," by Lawrence R. Rabiner, Marvin R. Sambur, and Carolyn E. Schmidt, IEEE Transactions on Acoustics, Speech, and Signal Processing, December 1975, makes use of a running median of 5 points followed by a liner smoother of 3 points with weights 1/4, 1/2, 1/4. The second filter acts to further reduce the effect of small-amplitude high-frequency noise. The excerpt below details these points.

> "Fig. 6 [4-1] shows a comparison between several alternative smoothing algorithms for an artificially created test input sequence. Fig. 6(a) shows the input sequence, and Fig. 6(b)-(d) show the outputs of a linear smoother (a 19-point finite impulse response (FIR) low-pass filter), a combination of median and linear smoother (a running 5 median and a 3-point Hanning window), and a median of 5 smoother, respectively. The smearing effects of the linear smoother at each input discontinuity are clearly in evidence in this figure, whereas the median smoother alone essentially preserves the data exactly. The combination of median and linear smoothing is seen to provide a good compromise between the median and linear smoothers in this example. Fig. 7 [4-2] shows the effects of adding broad-band noise to the input of Fig. 6. In this case, the median smoother is inadequate for filtering out the broad-band noise on the input, thereby producing a rough output sequence, as shown in Fig. 7(d). The linear smoother does an excellent job of filtering out the noise, as expected, and the output shown in Fig. 7(b) is almost identical to the output in Fig. 6(b) when there was no additive noise. Finally, the combination smoother is seen to again be a good compromise between the linear and the median smoothers. As seen in Fig. 7(c), the noise is smoothed a great deal, and the discontinuities in the input are fairly well preserved.

"In summary, a smoothing algorithm consisting of a combination of running medians and linear smoothing appears to be a reasonable candidate for smoothing noisy sequences with discontinuities."

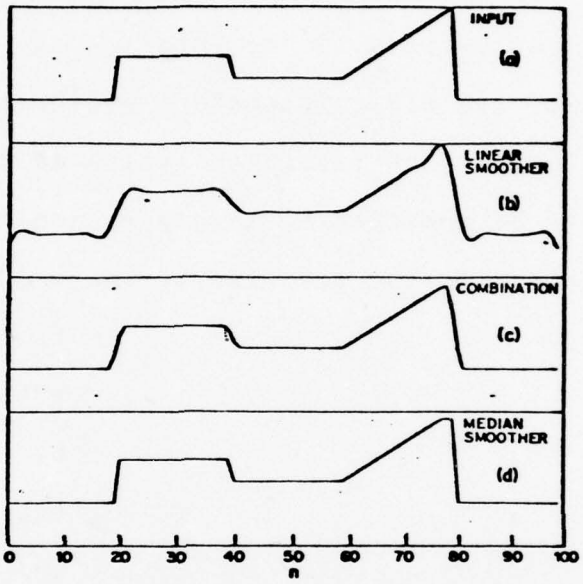


Fig. 4-1 Examples of several smoothed outputs for a simple test input.

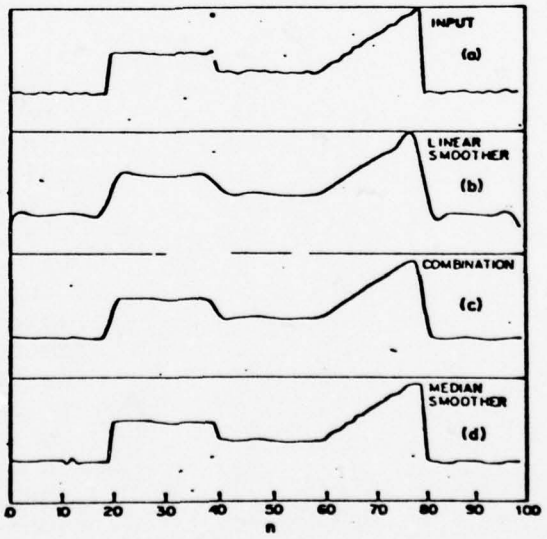


Fig. 4-2 Effects of additive noise on several smoothers

Computationally, such methods are quite efficient. Finding a running mean or median can be accomplished by maintaining a circular buffer of the n points along with a fill pointer. The mean can be calculated incrementally by subtracting the (weighted) value of the old point before overwriting it in the buffer, and adding in the (weighted) value of the new point. A running median can be computed by maintaining a heap structure, or a heap of pointers to the circular buffer.

4.3 Choosing a Smoothing Algorithm

The choice of a smoothing algorithm depends on two things: the properties of the raw data (i.e., packet delays), and the properties we want the smoothed data to have (e.g., predictiveness, responsiveness to certain sorts of changes in delay but not to others, etc.). As reported in Section 2, we have gathered data on the packet delays in the ARPANET. We have tested several smoothing algorithms by running them on this data. Our results and conclusions are reported in Section 4.3.1.

We must, however, enter a very important caveat about the results. As pointed out in Section 2, it seems that the characteristics of the raw data which we have gathered are extremely dependent on various characteristics of the current routing algorithm. If we implement a new routing algorithm whose line bandwidth and CPU bandwidth are much less than that of the current algorithm, the characteristics of the raw data may change a great deal. This may invalidate our choice of smoothing algorithm if our choice is based on data which we can gather at present. On the other hand, we can hardly base our choice on data which cannot be gathered until the future. As a result, we cannot expect that our initial choice of smoothing algorithm will be ideal for any new routing algorithm. When the new algorithm is implemented, we will have to continue to take measurements and test smoothers, so as to be able to refine our initial choice.

4.3.1 Results of Smoothing the Data

The purpose of smoothing the delay data is to get a measure which is somewhat predictive of future delay. Changes in the smoothed data should track major changes in the raw data fairly well, while being insensitive to minor changes which can be attributed to noise. There are a virtually unlimited number of possible smoothing algorithms which can be used, either individually, or in combination. It is impossible to fully test every conceivable smoother. After a good deal of initial trial and error experimentation, we settled on testing the following two smoothing techniques:

a) Median Smoothing. The rationale of median smoothing is discussed in Section 4.2. It is extremely sensitive to major discontinuities in the data, but insensitive to noise. The exact algorithm we tested is the following:

i) First, take a running median of 5 of the raw delay values.

ii) Second, quantize (round) the resultant point to the nearest 5 ms.

iii) Third, take a running median of 7 of the quantized points. (We added this step when it turned out that

the result of applying steps i) and ii) was insufficiently smooth.)

b) Block-average smoothing. The output from this smoother is simply the result of quantizing to the nearest 5 ms. the average of the first ten sampled points, followed by the average of sampled points 11-20, followed by the average of sampled points 21-30, etc. We chose a block average rather than a running average because we felt that the former would be more sensitive to discontinuities than the latter. We chose to average over a fixed number of packets rather than over a fixed time interval because we thought that our sampling frequency was too low to yield representative results for fixed-interval averaging.

We applied the two smoothers to the sum of the processing delay and modem queueing delay of each sampled packet. Some typical results are plotted in Figures 4-3 through 4-8. Figure 4-3 shows the sum of the processing delay and modem queueing delay for data from the line between ISI22 and ISI52. Figure 4-4 shows the result of applying the median smoother to that data, and Figure 4-5, the result of applying the average smoother. Figures 4-6, 4-7, and 4-8 are the corresponding plots for data obtained from the line between MIT6 and MIT44 during an experiment with artificially induced heavy load (multi-packet

messages) with reduced frequency of routing. Note that the median smoother produces one output point for every input point. In order to construct comparable plots with the averaging smoother, we have drawn the output as a constant during each measurement period; i.e., each averaged output is repeated ten times, once for each input point.

As can be seen from the plots, the average generally results in smoother output than the median does. Both the average and the median do a fairly good job of tracking the changes in delay, rising when delay rises, going down when delay goes down. Sometimes, however, the average will rise when the median goes down, or vice versa. It is these differences in the behavior of the two smoothers which we must evaluate in trying to decide between them.

The differences seem to be due to the following. A property of the median is that it eliminates individual points which differ greatly from the points which surround them in the sample. However, it does not eliminate small clusters of points which differ only slightly from the points surrounding them in the sample. For example, if most sampled points are around 10 ms., the median will eliminate an isolated point which has the value, say, 100 ms. It will not eliminate a cluster of 3 points which have the value 5 ms. The average, on the other hand, has just

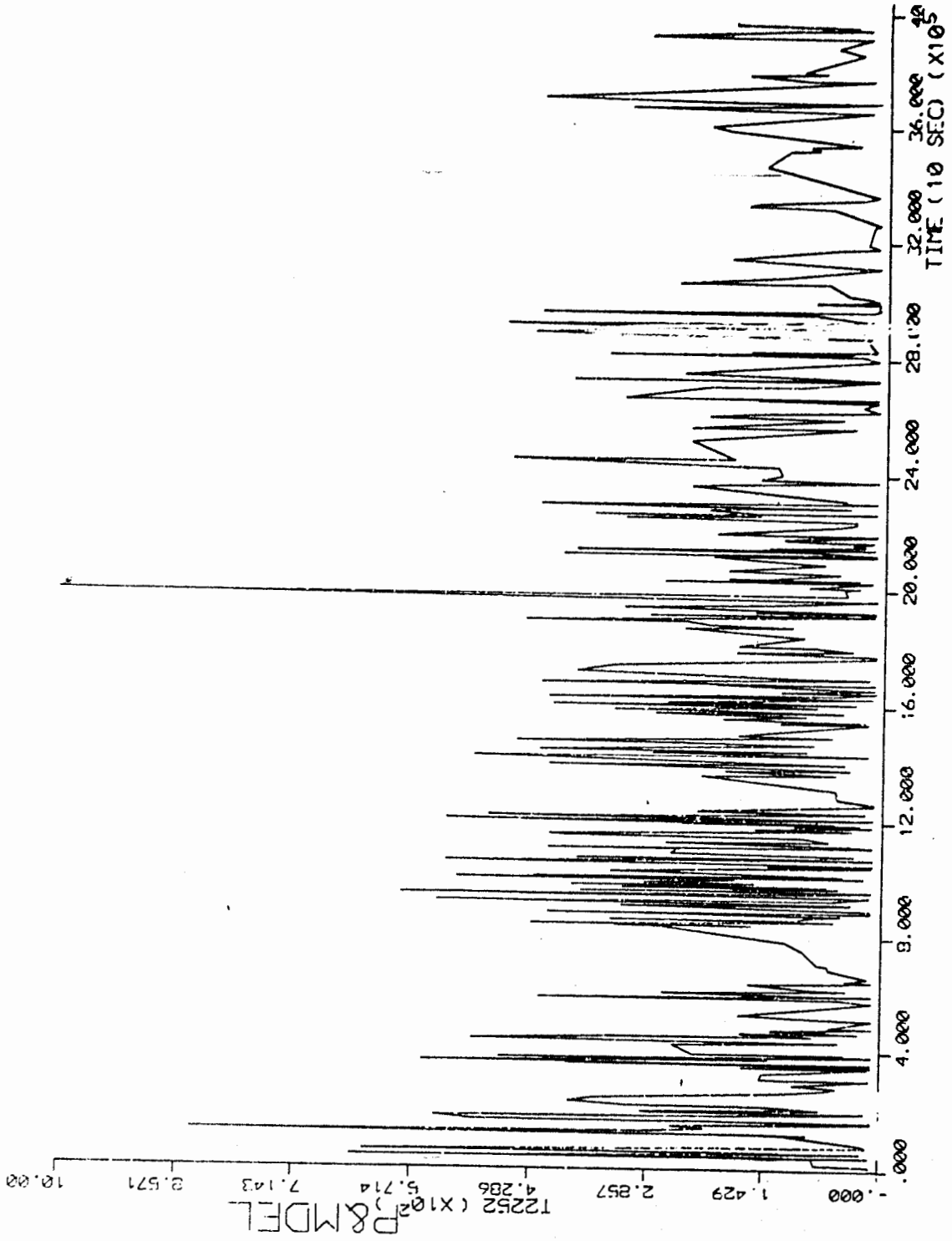


Figure 4-3 Processing Delay and Queueing Delays (Normal)

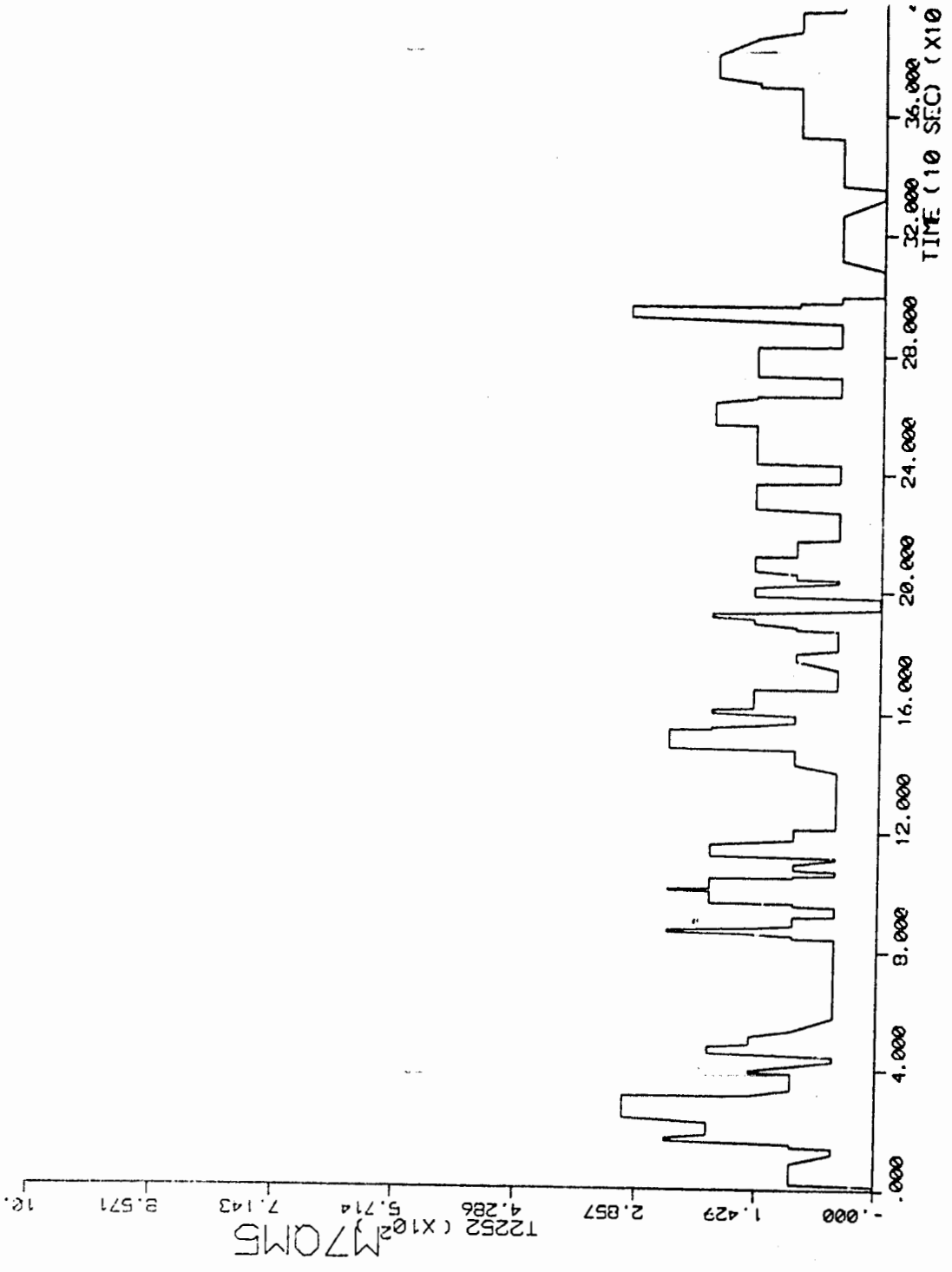


Figure 4-4 Median Smoothing (Normal)

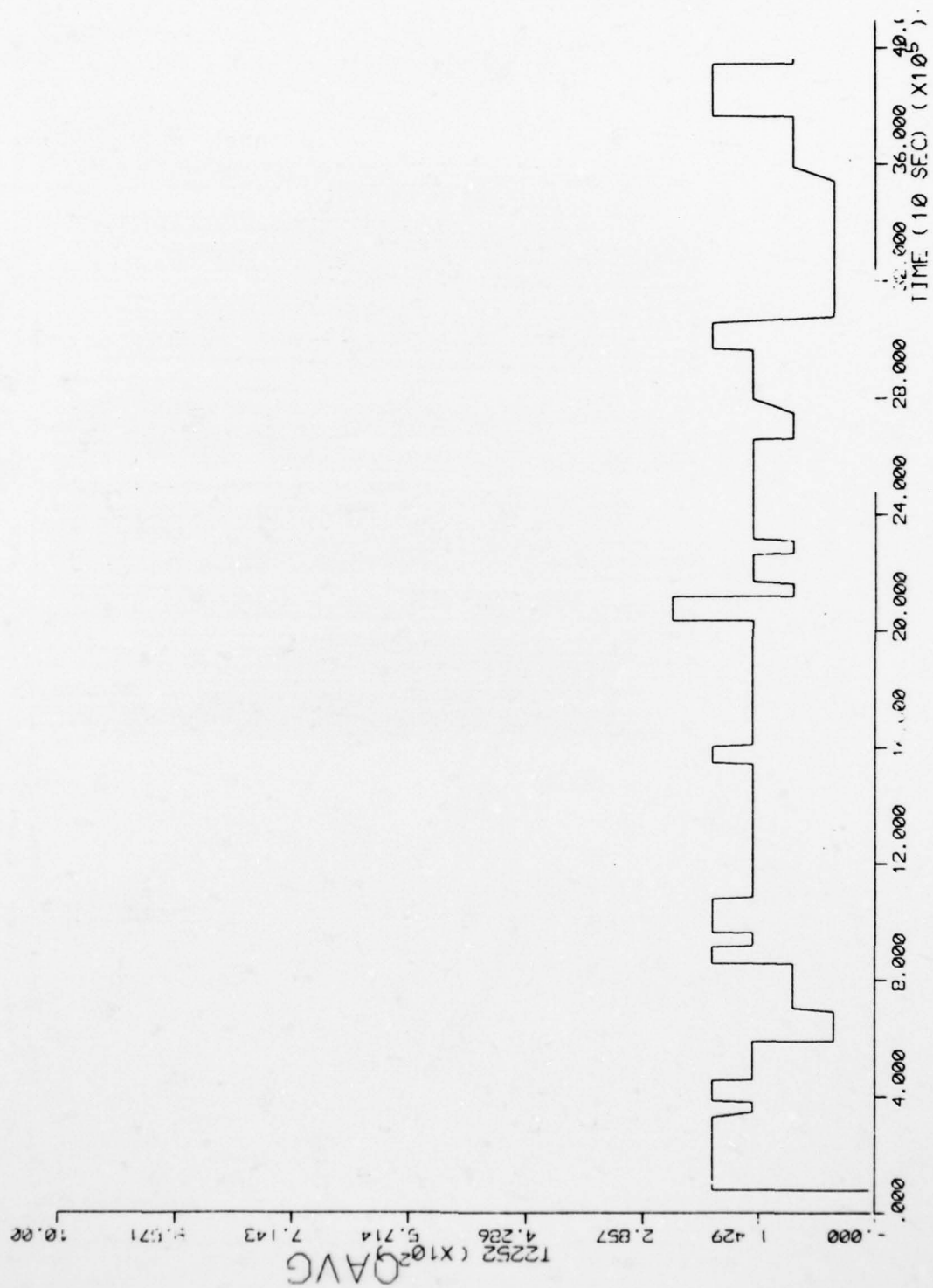


Figure 4-5 Quantized Block Averaging (Normal)

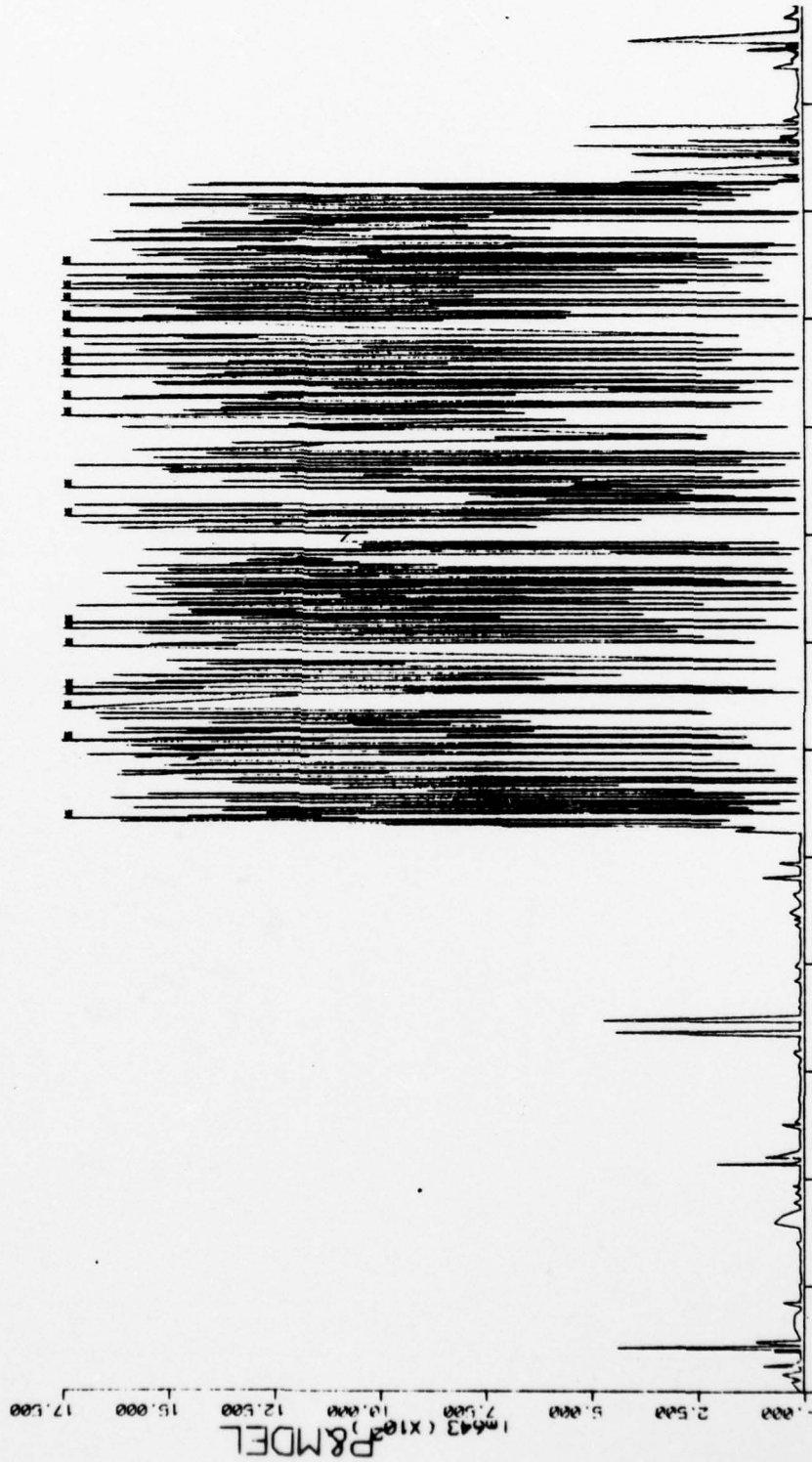


Figure 4-6 Processing Delay and Queueing Delays (Experimental)

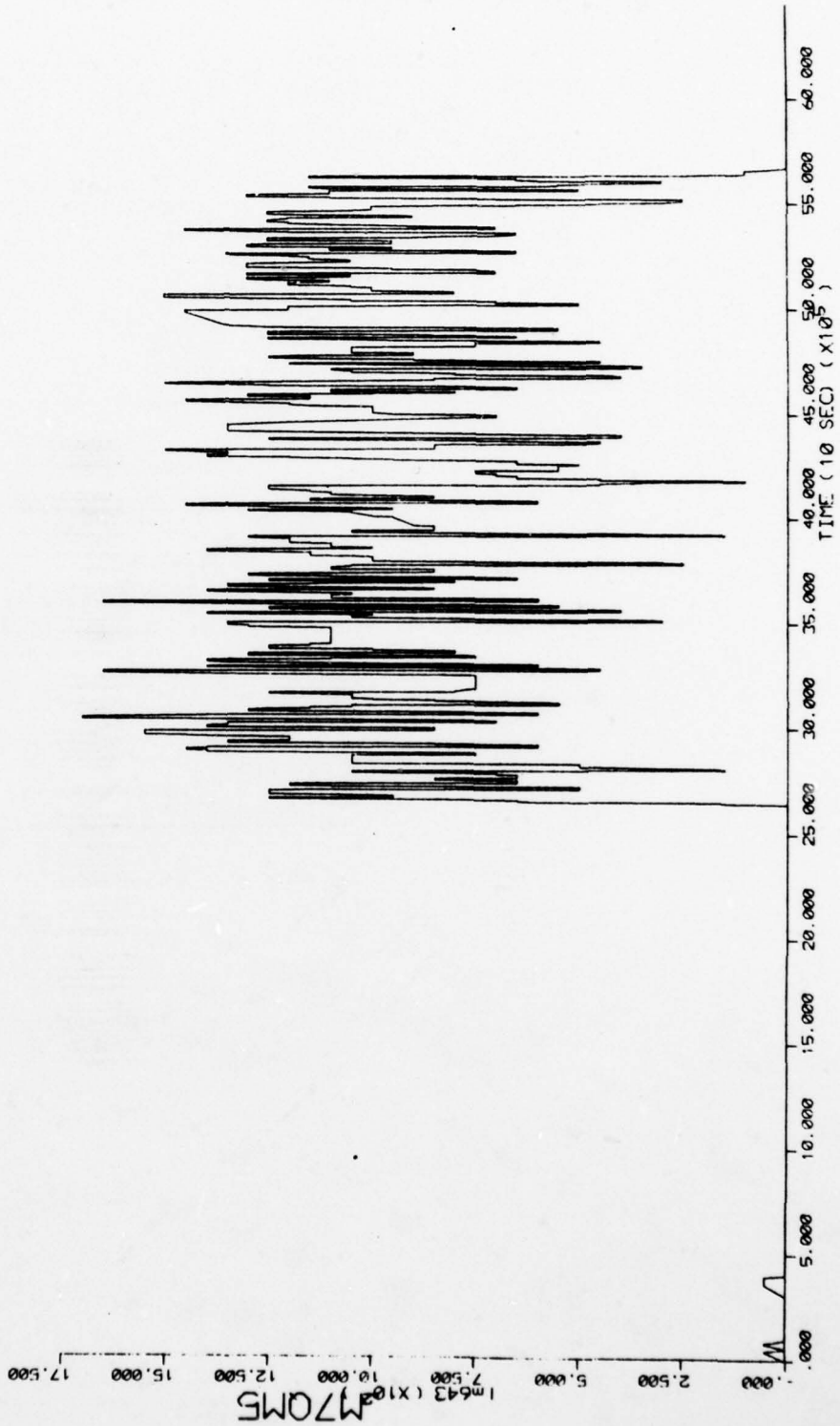


Figure 4-7 Median Smoothing (Experimental)

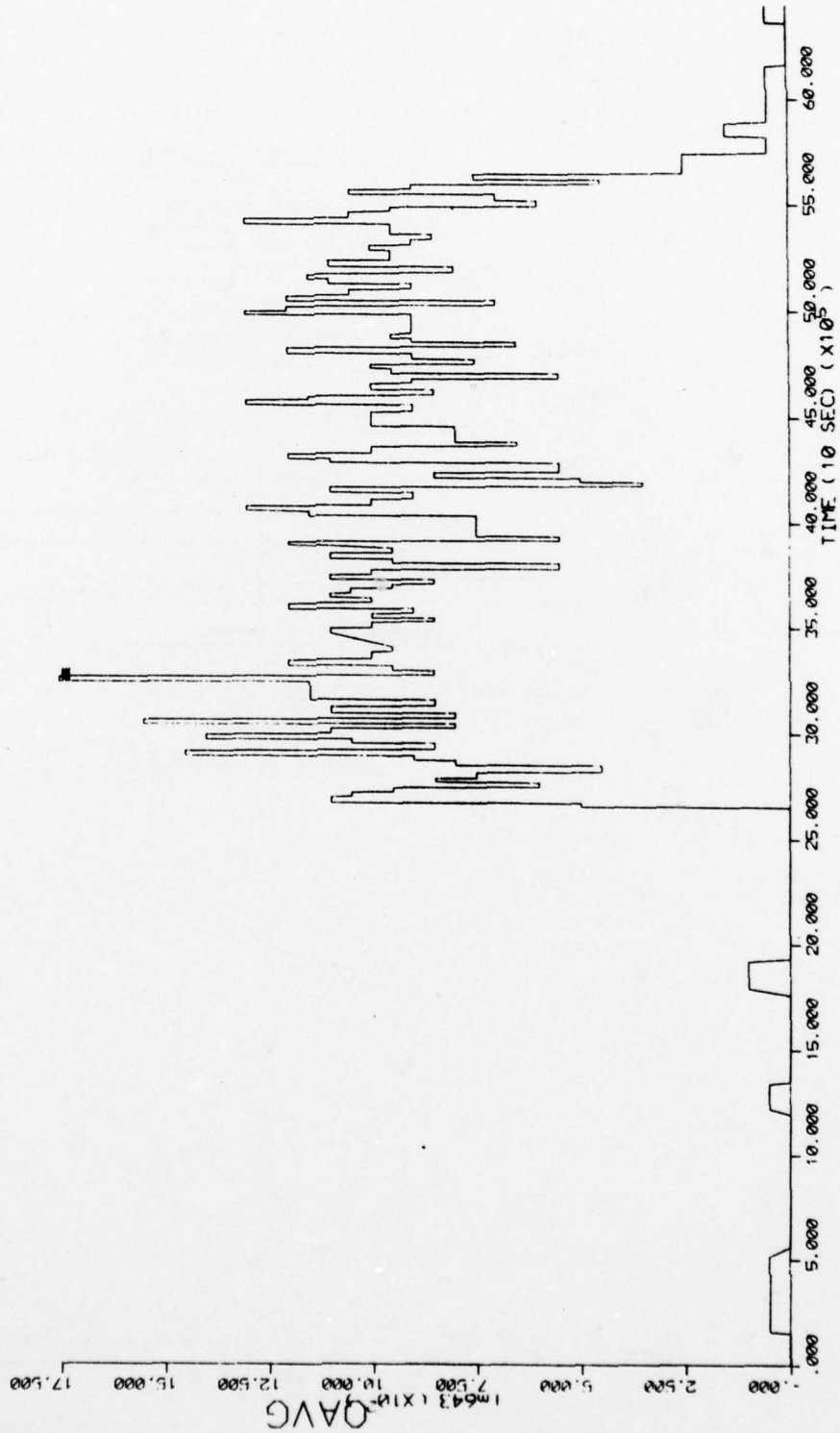


Figure 4-8 Quantized Block Averaging (Experimental)

the opposite properties. It will be greatly affected by a single point with the value 100 ms., but much less affected by 3 points with the value 5 ms. The cases where our median and average smoothers seem to disagree can be attributed to these properties.

We believe that where the average and median smoothers disagree, the average is preferable, for the following reasons:

a) A single point which is extremely anomalous in the sample may not be so anomalous in the data. It may be that where we see one 100 ms. point in the sampled data, there are three or four of them in the real data. If this is the case, then we want to count that point, as the average does, not eliminate it, as the median does. (Note, however, that the reason may be biased by the limitations of our data gathering methodology, see Appendix 1.)

b) A small cluster of slightly anomalous points should not result in the generation of a routing update. Thus a function like the average, which is not sensitive to such clusters, is better than a function like the median, which is. Of course, we need not report a change to the network just because our smoother detects it, but we might as well try to get a smoother which does not react to changes which we would not want to report anyway.

c) While the median is better than the average in filtering out certain kinds of noise, it is not clear that we can identify any of our data as noise.

d) It is true that the average is unlikely to react as fast as the median in detecting major discontinuities. But,

i) A comparison of Figures 4-7 and 4-8 shows that the average can also detect discontinuities reasonably well.

ii) The sort of major discontinuities which may be important to detect quickly does not seem to occur very often. We have seen them in our data only under artificially induced load.

We therefore intend to implement a block average smoother initially. However, if it should turn out that the characteristics of delay change a lot when the new routing algorithm is implemented, we may need to re-evaluate this choice.

4.4 Implementing Delay Measurement and Smoothing in the IMP

The delay measurement portion of a new routing algorithm should be implemented in four separate modules: a sampling module, a smoothing module, a comparing module, and a reporting module. The sampler would be responsible for sampling the delay values of packets being transmitted on a particular line. The output of the sampler would be input to the smoother, which would implement an algorithm such as the ones discussed in Section 4.3. The output of the smoother would be input to the comparer. The comparer will compare the smoother's latest output with previous ones, and decide whether to generate a routing update message. The actual generation and transmission of a routing update would be the responsibility of the reporting module.

There are still many unresolved questions about the algorithms to be used by each of these modules, and there are many experiments which must be done in order to answer these questions. We believe that we have reached the limits of what we can learn using our present means of gathering data. The only way to gather the additional data needed to answer our remaining questions is to implement these four modules and run them in the IMPs. Initially, we will use the modules only for generating data, under a variety of normal and/or experimentally created conditions. Eventually, the modules will become an essential part of a new routing algorithm.

5. POSSIBLE IMPROVEMENTS TO CURRENT ARPANET ROUTING ALGORITHM

This section examines three possible improvements to the current ARPANET routing algorithm:

-- event-driven updating
-- improved reachability determination (faster and more efficient)
-- improved loop suppression

We were able to develop improved techniques in each of these aspects of the routing algorithm. However, some problems still remain, and we were led to consider a completely new routing algorithm (SPF, described in Section 6 below) which does not suffer from these deficiencies.

5.1 Updating Policy

An alternative to the current use of periodic routing updates in the ARPANET is to use event-driven routing updates. If this technique is applied to the present ARPANET algorithm, a node would transmit routing messages to its neighbors only if its routing table changes or if its delay to some destination changes by some pre-specified amount. Such a change can come about either because the delay on one or more of the node's own lines changes or because of a routing message received from a neighbor. With event-driven updates it may still be necessary to transmit routing messages periodically, but the frequency of these messages will be quite low.

The principal motivation for using event-driven updating is that routing should adapt very quickly to changes. Furthermore, since a given routing change often affects only a fraction of the nodes in the network, and since the frequency with which changes occur might under normal circumstances be relatively low, we can anticipate that the average line and node bandwidths required by this technique would both be lower than that of the present algorithm. Finally, the size of the routing message itself could possibly be reduced since information only on the changed routes need be circulated.

At present there are two potential problems that we see with such event-driven updating. First, when a significant routing change occurs, there may be some appreciable settling time before new and stable routes are obtained. During this settling time, many nodes are exchanging information which is only partially up-to-date. The exchange of routing messages cannot end until all nodes have all the latest information, and this can require that many routing messages be exchanged. In other words, the necessary peak line and node bandwidths required may be excessive; at any rate, the peak/average ratio will be very high. A possible solution to this problem is to set a minimum updating period for each node, with no more than one update per period. Clearly, if this period is chosen to be 2/3 of a second, then the

adaptation time would be similar to that of the present algorithm (ignoring hold-down). A very short period would permit rapid adaptation but could cause the previously mentioned excessive use of bandwidth. Some thought is required to determine whether such an updating period is beneficial, and if so, the value of the period that yields a good compromise between speed of adaptation and elimination of unnecessary and unreliable updates.

A second problem with event-driven updates is somewhat more involved and requires more detailed explanation. Assume first that on the basis of a routing update, node A's delay to node B decreases significantly. Then, whether or not A was formerly using this route to B, A will now use the improved route and since its delay table changes, A will forward updates to its neighbors. Now assume that A receives information that indicates that the delay to B via line L is larger than A's present delay to B. If A's present route to B is not via L, then A simply discards the message. If, however, A is using L, then A must update its table and forward the change to its neighbors; but A has no information on alternate routes, and, therefore, it must still route messages to B via L, even though the delay along this path may be excessive and better paths might exist. Thus, the basic event-driven updating allows information on routing problems to propagate quickly, but it does not provide a mechanism to alleviate these problems.

We visualize two possible solutions to this problem. As one alternative, each node could maintain information on a second-best route to each destination. With this approach additional storage is required at each node, but no additional line bandwidth is needed, since this information is contained in the routing messages that are normally exchanged. However, there may be some difficulties associated with maintaining two sets of up-to-date routing and delay tables and with ensuring that the delay along the second-best path is not also affected.

A more desirable alternative is for a node to request updates from its neighbors. In the above example in which the delay to B increases, node A would request updates from all its neighbors except the one on line L. In fact, A could incorporate this request in the routing updates it transmits to those neighbors. The overall scheme might thus function as follows. As above, we assume that node A receives an update on its route to node B via line L.

1. If the delay to B decreased, then A updates its tables and forwards the message on its other lines but does not request an update from these neighbors.

2. If the delay to B increased, then if A was using line L, it updates its tables, forwards the information on its other lines, and requests updates from these neighbors.

3. If, along with the routing update for line L, node A receives an update request, then A updates its tables and services the request unless the update caused A's

delay via L to increase. In this case A ignores the update request.

Step 3 above insures that unreliable information will not be exchanged between neighboring nodes. The technique works by propagating information on increased delays outward along with update requests and by having these requests ignored until the increased delay no longer affects a node; this node then sends information on a superior path. Of course, before the proposed technique is adopted, considerable thought must be devoted to insuring that loops do not form.

5.2 Improved Reachability Determination

The present ARPANET routing algorithm determines reachability by means of the following heuristic: whenever the hop-count to a certain destination becomes "infinite" (i.e. 32) and remains infinite for 10 seconds, the destination is declared unreachable. Because the ARPANET routing algorithm does not maintain information about individual network lines, and because it cannot distinguish among paths which have the same "next hop", there is probably no way it can determine reachability other than by using some such heuristic. That is, the algorithm is not capable of providing more information about a particular destination than whether or not there is a path to it, and if so, what the hop-count and delay along that path are. Therefore, there is little that can be done to determine that an IMP is unreachable other than to note that there is no path to it, and that there has not been any path to it for a certain time interval. As a result, the only way to improve the reachability aspect of the ARPANET routing algorithm is to improve the performance of the heuristic.

There are two obvious problems with the present performance of the reachability heuristic: It does not work quickly enough and it uses an excessive amount of line bandwidth. These points are discussed in the sections below.

5.2.1 Speed of Adaptation

No IMP can be declared unreachable until 10 seconds after it actually becomes unreachable. This means that there is a period of at least 10 seconds during which the network does not know what to do with packets for an unreachable destination. During this period, the network is forced to buffer those packets in store-and-forward space. Since the network does not have enough store-and-forward capacity for such buffering, the result is that packets have to be discarded, and the packets which get discarded may not even be packets for the IMP which is unreachable.

The optimal length of the time interval during which packets for possibly unreachable IMPs must be buffered depends on the speed at which the network can find a second path to a destination when the original path to that destination is lost. If the time interval is too small, IMPs will be spuriously declared unreachable, resulting in packet loss; if the interval is too large, packets are also lost. We have data which show that an interval of 4 seconds would be too short. However, since packets can be discarded from the subnet after being buffered in store-and-forward space for 4 seconds, any interval that exceeds 4 seconds is too long.

If the ARPANET routing algorithm were modified to use the event-driven updating scheme described in the previous section, the time interval to discover an alternate path would of course become much shorter. Hence the reachability determination would automatically be improved, since the time interval could be reduced from 10 seconds to (hopefully) less than 4. However, it is difficult to say a priori what the interval would have to be with event-driven updating.

5.2.2 Bandwidth Considerations

Every routing update message contains a hop-count, as well as a delay for each destination. The hop-count, which is used only to determine reachability, adds 5 bits per destination to each update. Since the hop-counts change very rarely (relative to the update frequency), this wastes a great deal of line bandwidth. Thus, it would be useful to remove the hop-counts from the routing update messages. If the hop-counts are removed from the routing update messages, there are two ways to determine reachability:

a) An IMP may be declared unreachable when the delay (rather than the hop-count) to it becomes infinite and remains infinite over a certain time interval. This way of determining reachability dispenses with the hop-counts entirely.

b) Hop-counts may be contained in "reachability update messages" which are separate from the routing update messages. Reachability updates would be event-driven, sent only when there is a change in the hop-count to some destination. These updates would be generated very rarely.

The method of dispensing with hop-counts entirely was at one time considered by the ARPANET designers, and rejected. Their reason for rejecting it was valid at the time, but may not be as applicable with hold down or other loop-suppression (see Section 5.3 below) and event-driven updating as part of the routing algorithm. Their reasoning was as follows:

> Let IMPs A, B, C, D, and E be connected in a chain, and suppose C crashes. Then C's neighbors, B and D, will immediately set their delays to C to be infinite. But if B and D now get updates from their respective neighbors A and E, which indicate (because their information is outdated) that they have finite delays to C, B and D will assume that they too have finite delays to C. Thus the infinite delays are purged from the network. Of course, if C is really unreachable, further updates will eventually cause the delay from all

other IMPs to get larger and larger, eventually reaching its maximum value. But since the maximum value of delay is so large, this will take an excessively long time. So long as the maximum value of hops is much smaller than the maximum value of delay, using hop-counts results in a much quicker determination of reachability.

However, if there is event-driven updating and loop-suppression, there will be less outdated information in the network, and much of it will be ignored anyway. Therefore, once the delay to a certain destination is set to infinity, the probability is high that it will remain at infinity, unless an alternate path really does exist. However, the probability is not 100%, since routing updates about the same event can reach an IMP from different directions at different times (as detailed in BBN Report 3641). Therefore, there are some cases where the reachability determination will be unacceptably slow if it is based on delay counting up to infinity. Furthermore, when such cases do occur, they will cause an excessive number of routing update messages to be generated.

An event-driven routing algorithm with loop-suppression would need hop-counts much less frequently than the unmodified algorithm. However, hop counts are safer to use for reachability

determinations than are delay values. The question that must be resolved is whether the additional cost involved in using hop-counts is justified on the basis of the additional amount of safety they provide.

5.3 Improved Loop Suppression

One important way of improving the ARPANET routing algorithm would be to replace the current means of loop-suppression (hold-down) with a more effective one. In this section, an alternative method of loop-suppression is proposed. Section 5.3.1 compares it to hold-down from a theoretical point of view. Section 5.3.2 discusses implementation considerations.

5.3.1 Hold-Down vs. Explicit Information

Hold-down (HD) is a set of procedures added several years ago to the basic ARPANET routing algorithm. Its purpose is to suppress the formation of path loops, and thereby to decrease the time it takes for the network to adapt to changes in topology or line-loading. HD will be compared throughout this section with a different loop-suppression scheme which we term the "Explicit Information" scheme (EI). We consider first the issue of suppressing loops of arbitrary length. Later we consider the issue of suppressing only loops of two nodes ("ping-pong loops").

Suppose IMP A has a non-looping path to some destination IMP X. In order for a looping path to form, the following two conditions must hold:

1. A receives a routing update from one of its neighbors (say, from B) according to which the delay from A to X via B is less than A's current delay to X.

2. A is included in B's path to X. (If a network path is represented as an ordered sequence of IMPs whose first element is the source, whose last element is the destination, and is such that any contiguous pair of IMPs in the sequence are topological neighbors, then we say that an IMP is included in a path if it appears in the sequence which represents the path. A looping path is a sequence in which some IMP is included more than once.)

If condition 1 does not hold, a looping path is not formed, because A does not switch paths. If condition 2 does not hold, A may switch paths, but the new path will not have a loop. Note that in the basic ARPANET routing algorithm, conditions 1 and 2 are not only necessary for loop-formation, but are sufficient for it. One way of suppressing loops then is to detect the co-occurrence of conditions 1 and 2, and to take appropriate action on that basis. A very simple scheme is the following:

Explicit Information Scheme (EI): When a node sends a routing update message to a neighbor, it specifies the path to

each destination. When a node processes a routing update from a neighbor, it checks to see whether it is included in the neighbor's path to a destination. If so, it acts as if the neighbor had reported infinite delay to that destination.

Of course, EI could hardly be implemented in a real network. It is not practical to have the routing update messages contain the entire path from a node to all others. It is mentioned here only because it is a simple, "brute-force" method of loop-suppression which can be interestingly contrasted with HD. It is clear that EI prevents many loops from forming, since an IMP will never switch to a path which has infinite delay. However, EI does not prevent all loops. Suppose that B has a path to X which does not include A. It is possible that B sends a routing update to A, and immediately afterwards switches to a path which does include A. In this case, it is possible that A is included in B's path to X at the time A processes the routing update from B, although it was not so included at the time B sent the routing update to A. That is, EI cannot ensure that A will always know whether condition 2 holds at the time A's routing calculation is being done. Thus EI does allow some loops to form. However, EI does ensure that any loop which forms will be quickly broken, and a non-looping path quickly established to replace it. For, once a loop does form, EI causes the IMPs to see

infinite delay on the looping path. This will cause a non-looping path to be preferred as soon as one can be found.

To sum up: EI works by enabling the IMPs to detect conditions which are sufficient for the formation of loops. In many cases, the information can be used to prevent loops from forming. In other cases, the information can be used to eliminate loops, and to ensure that they are replaced with non-looping paths. EI's major disadvantage is that a pair of neighboring IMPs must work together to determine whether conditions sufficient for loop formation are present. This requires very large routing update messages.

Hold-down is an attempt to suppress loops by utilizing purely local information, thereby avoiding EI's major disadvantage. To understand how it works, we need the notion of one routing update message being based on another. If A and B are neighboring IMPs, and a and b are routing update messages generated by A and B respectively, then a is directly based on b with respect to destination D if and only if b is the most recent update from B which A processed before generating a, and a reports the delay to D for a path whose next hop after A is B. If P and Q are any two IMPs, not necessarily neighbors, and p and q are routing update messages generated by P and Q respectively, then p is based on q with respect to destination D if and only if

either p is directly based on q with respect to D, or there is a third routing update message r such that p is directly based on r with respect to D, and r is based on q with respect to D.

With this notion, we can present the following two propositions (again, let A and B be neighbors, and let X be some destination IMP):

3. If conditions 1 and 2 hold, then the update which A most recently received from B must be based on an old update generated by A and sent to B, and furthermore, A's delay to X must have increased since that old update was generated.

4. Let p and q be routing update messages generated by some pair of IMPs, not necessarily neighbors. Let t(p) and t(q) be the times at which p and q were generated respectively. Then there is some length of time T such that:

   if t(p) - t(q) >= T, then p is not based on q

Proposition 3 is fairly obvious. B could not ever switch to a path which includes A, except as a result of receiving an update from A, or else receiving from another neighbor an update which is based upon an update received from A. And B could not see a

smaller delay on a path including A than is seen by A itself, unless A's delay has increased since it generated the update on which B's information is based. Proposition 4 simply states that the information contained in any given routing update has only a finite life-span in the network, which is less than the value T. These two propositions suggest a purely local scheme for loop-suppression:

Hold-down scheme (HD): Whenever a node's delay to a particular destination increases, it pays no attention to what its neighbors say about that destination until an interval of length T elapses.

If propositions 3 and 4 are true, HD ensures that whenever a routing update message is received such that conditions 1 and 2 hold, the update will be ignored. That is, HD ensures that the only updates which are processed are those for which conditions 1 and 2 do not hold. Since conditions 1 and 2 are necessary for the formation of loops, HD ensures that no loops are formed. In fact, HD is more effective in preventing loops than is EI. Whereas EI allows some loops to form temporarily, HD never allows a loop to form. EI works by detecting sufficient conditions for the formation of loops. Despite the large overhead incurred in detecting these conditions, it is not always able to detect them soon enough to prevent a loop from forming (although it does

detect them soon enough to be able to to break loops quickly). HD, on the other hand, works by detecting conditions which are necessary for the formation of loops. It is always able to detect these conditions in time to prevent the formation of loops, and it detects them by using purely local information, thereby incurring low overhead.

Nevertheless, HD has some serious disadvantages. For one thing, in a network the size of the ARPANET, the value of T is on the order of minutes. This means that whenever the delay between A and X increases, the network will be very slow to adapt. Since the purpose of a loop suppression scheme is to speed up adaptation, not slow it down, HD may be self-defeating.

Another disadvantage stems from the fact that although propositions 3 and 4 are true, their converses are false. The conditions which HD detects may be necessary for the formation of loops, but they are not sufficient. This means that although HD causes the IMPs to ignore updates for which conditions 1 and 2 hold, it also causes them to ignore, unnecessarily, some updates for which conditions 1 and 2 do not hold. In fact, for all that has been said so far, it could well be the case that most of the updates which HD causes the IMPs to ignore are updates which, if processed, would not result in loops. If this were the case, then HD's major effect would be to slow adaptation time, rather

than to speed it up. So HD's utility depends on the putative fact that most of the updates it causes to be ignored would result in loops if they were processed. That is, HD's utility depends on the assumption that the conditions it detects, which are necessary for loop formation, are also highly correlated with conditions which are sufficient for loop formation. This means that HD is a heuristic, probabilistic procedure, whose true utility cannot be determined a priori.

While neither EI nor HD appears to be a practical way of suppressing loops, they are perhaps too ambitious in that they seek to suppress loops of all lengths. It must be noted however that without any loop suppression technique, long loops are much less likely to form than are short loops. Since each hop in a path adds appreciably to the total delay on that path, paths with long loops will generally show a much higher delay than paths with short loops, or no loops at all. Therefore, the longer a loop is, the less likely it is that conditions 1 and 2 can be satisfied simultaneously, hence the less likely it is that the loop can actually form. This means that a scheme which only suppresses, say, two-node "ping-pong" loops, may be nearly as effective at suppressing loops as a scheme which tries to suppress all loops. Both EI and HD can be easily modified to attempt to suppress only "ping-pong" loops.

To modify EI, we need only note that in order to suppress loops which are n nodes long, it is only necessary to inform one's neighbor of the next n-1 hops in one's path. Therefore, the following scheme suffices.

Explicit Information Scheme for 2-node loops (EI2): When a node sends a routing update message to a neighbor, it specifies, for each destination whether that neighbor is the next hop on its path to that destination. When processing a routing update message from a neighbor, it checks to see if it is that neighbor's next hop to any destination. If so, it acts as if that neighbor had reported infinite delay to that destination.

EI2 is a more practical scheme than is EI. If it is allowable to send different routing messages to different neighbors, then EI2 can be implemented with no additional bandwidth at all; if the same routing message must be sent to all neighbors, the cost is still low: only n bits per destination per routing update, where n is one more than the greatest integer of the base 2 logarithm of the number of neighbors an IMP may have. These two means of implementation are discussed in Section 5.3.2. Otherwise, EI2 is just like EI - it cannot always prevent loops, but it quickly eliminates loops which do form. And it never causes a routing update message to be ignored unnecessarily.

To see how to modify HD so as to prevent only two-node loops, we note that in order for a two-node loop to form between IMPs A and B for destination X, the following two conditions must hold:

1'. A receives a routing update from B according to which the delay from A to X via B is less than A's current delay to X.

2'. A is the next hop on B's path to X.

Furthermore,

3'. If conditions 1' and 2' hold, then the update which A most recently received from B must be directly based on an old update generated by A and sent to B, and furthermore, A's delay to X must have increased since that old update was generated.

4'. Let a and b be routing update messages generated by some pair of neighboring IMPs. Let t(a) and t(b) be the times at which a and b were generated, respectively. Then there is some length of time T', which is much smaller than the T of proposition 4, such that:

if t(a)-t(b)>=T', a is not directly based on b

Therefore, the following scheme will work:

Hold-down Scheme for Two-node Loops (HD2): Whenever a node's delay to a certain destination increases, it pays no attention to what its neighbors say about that destination until an interval of length T' elapses.

Since T' is very much smaller than T, HD2 is not as impractical as HD. But HD2 shares many of its properties with HD. Like HD, HD2 prevents all two-node loops. But it does it at a cost of causing the IMPs to unnecessarily ignore many routing update messages, thereby slowing, rather than speeding, the convergence of the basic routing algorithm in many cases.

HD2 is not the scheme in use in the ARPANET. Rather, the ARPANET uses a peculiar scheme which we will call "HD2-8".

HD2-8: Whenever a node's delay to a certain destination increases by at least 8 units since it last sent a routing update message to some neighbor, it pays no attention to what its neighbors say about that destination until an interval of length T' elapses.

It is clear that HD2-8 has the virtue of not causing the IMPs to ignore routing updates as often as they will if HD2 is used. But HD2-8 has a rather serious disadvantage -- not only

does it fail to prevent some loops, but when loops do form, they get locked up. (See Section 2). This would seem to rule it out as a practical loop-suppression technique.

If loop-suppression were an end in itself, HD2 would be preferable to EI2, since the former always prevents loops from forming, while the latter does not. However, if loop-suppression is merely a means of speeding the convergence of the basic routing algorithm, EI2 seems preferable, since it never causes routing update messages to be ignored unnecessarily.

5.3.2 Loop Suppression as Part of an Improved ARPANET Routing Algorithm

The main purpose of a loop-suppression scheme in the ARPANET would be to prevent the occurrences of long-lived "ping-pong", or 2-node, loops. A look at the current ARPANET logical map shows that three-node loops, four-node loops, and five-node loops are impossible, since the topological pre-conditions for them do not exist. That is, there is in the ARPANET no set of three nodes connected in a triangle, no set of four nodes connected in a rectangle, and no set of five nodes connected in a pentagon. There are several sets of six nodes connected in a hexagon, though, so loops of six or more nodes are possible. However, the probability of a loop's occurring is inversely related to the

diameter of the loop, while the cost of a loop-suppression scheme is directly related to the diameter of the loops it can suppress. Therefore, schemes capable of suppressing loops of six or more nodes are probably not worth considering, which means that only the suppression of ping-pong loops, is of any importance in the ARPANET.

Any alteration to the ARPANET routing algorithm which tends to stabilize the estimates of delay will automatically help to prevent loops, since it will make spurious routing changes, and hence loops, less likely to occur. Similarly, any alteration which causes new information to spread faster and more uniformly through the network will automatically help to prevent loops, since it will make it less likely that nodes disagree about the delays to various destinations. Therefore, both the event-driven updating and the improved delay estimates described in previous sections would help to make loops less likely. However, it does not seem likely that these improvements would totally obviate the need for an explicit loop-suppression scheme. In the remainder of this section, we describe two variants of the EI2 scheme, each of which seems implementable in conjunction with the other suggested improvements to the ARPANET routing algorithm.

1. In this variant, a separate routing update message must be sent to each neighbor. Let I be the IMP which is sending the

update, and K the IMP to which it is being sent. The set of possible destinations can be divided into two disjoint subsets -- those for which K is the next hop after I, and those for which K is not the next hop after I. For destinations in the former class, the update should specify infinite delay, rather than the actual estimated delay. This will prevent K from routing to I packets which would only be routed back to K.

2. In this variant, the same routing message is sent to each neighbor. The sending IMP indicates, in addition to the delay to each destination, the next hop on its path to that destination. This can be done in three bits, providing that each IMP has its neighbors' neighbor tables, and that the bit-coding limitation on the number of neighbors is acceptable. When the receiving IMP notes that it is the next hop on the sending IMP's path to some destination, it acts as if the sending IMP had reported infinite delay to that destination. (Note that the update message in this second variant also contains all the information needed to suppress three-node loops.)

While this loop-suppression scheme does not prevent all ping-pong loops, it does ensure that any ping-pong loops which do form will be very short-lived. This would be especially true if it were implemented in conjunction with the other improvements described earlier in Sections 5.1 and 5.2.

6. SHORTEST PATH FIRST ALGORITHM

6.1 Introduction

Many algorithms have been devised for finding the shortest path through a network. A recent survey article [1] discusses some of these algorithms and also references several other survey articles. The basic algorithm we shall consider is attributed to Dijkstra [2]; because of its search rule, we call it the shortest path first (SPF) algorithm. The following section first describes this basic algorithm which is used to initially generate the shortest path tree and then explains the important additions we have developed for modifying the tree if network changes occur. Section 6.3 presents some analytic results for predicting SPF running times, and some quantitative results obtained by running the algorithm in FORTRAN on TENEX, and on the 316 and Pluribus. Section 6.4 discusses some basic issues on how updates (i.e., information regarding line and node status) may be handled. Section 6.5 compares the SPF algorithm with the present algorithm. The final section contains the conclusions from our work with SPF.

6.2 Shortest Path First Algorithm (SPF)

6.2.1 Basic Algorithm

The basic algorithm for finding the shortest path tree from a given source node is a way of building up the tree node by node. That is, the tree initially consists of just the source node. Then the tree is augmented to contain the node that is closest to the source and that is adjacent to a node already on the tree. The process continues by repetition of this last step. The tree is built up shortest-paths-first -- hence the name of the algorithm. Eventually the furthest node from the source is added to the tree, and the algorithm terminates. As a by-product of the algorithm, it is easy to produce an ARPANET-like routing table. Figure 6-1 presents an example shortest path tree for the ARPANET, to aid in visualizing the operation of the algorithm.

In the ARPANET, each IMP would run the algorithm with itself as the source. In order to run the algorithm each node must maintain a data base representing the topology of the network. A key component in the data base is the "length" of every line in the network (where "length" is not physical length, but rather some relevant metric such as delay).

We begin with a semi-formal description of the algorithm, followed by more verbose commentary. Let SOURCE be the node in

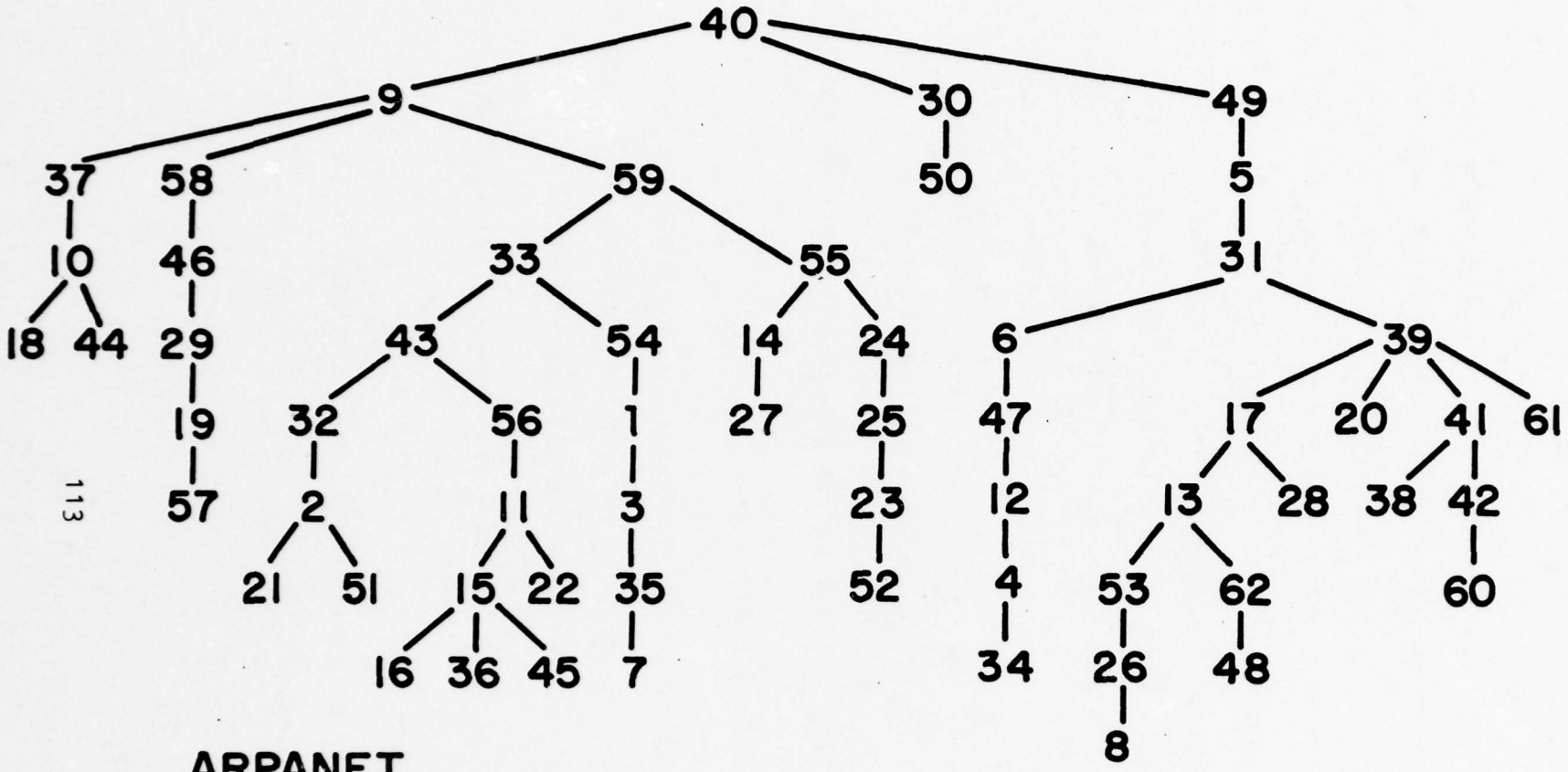


Figure 6-1 Example Shortest Path Tree

which the algorithm is running. The algorithm's basic data structure, LIST, is a variable-length list whose elements are ordered triples. An ordered triple T is of the form <SON, FATHER, DISTANCE>, where SON and FATHER are nodes, and DISTANCE is a number. (We use the notation SON(T) in the obvious way to mean the first element of the triple T.) Each triple represents a particular path from SOURCE to SON. The penultimate hop on this path is FATHER, and the total "length" of this path is DISTANCE.

To initialize the algorithm, place <SOURCE,SOURCE,0> on LIST. The algorithm itself consists of the following steps:

1. Search LIST for the triple T with the smallest DISTANCE.

2. Remove T from LIST.

3. Place SON(T) on the shortest path tree so that its father on the tree is FATHER(T). [Exception: If SON(T) = SOURCE, place it in the tree as its root]

4. For each neighbor N of SON(T), do <u>one</u> of the following steps:

   a. If N is already in the shortest path tree, do nothing.

b. If there is no triple T' on LIST such that SON(T') = N, then place the triple <N, SON(T), DISTANCE(T) + LINE_LENGTH (SON(T), N)> on LIST.

c. If there is already a triple T' on LIST such that SON(T') = N, and if DISTANCE(T') $\leq$ DISTANCE(T) + LINE_LENGTH (SON(T), N), do nothing.

d. If there is already a triple T' on LIST such that SON(T') = N, and if DISTANCE(T') > DISTANCE(T) + LINE_LENGTH (SON(T) N), then

i) Remove T' from LIST

ii) Place the triple < N, SON(T), DISTANCE(T) + LINE_LENGTH (SON(T), N) > on LIST

5. If LIST is non-empty, go to 1. Otherwise, the algorithm is finished.

Commentary

Remember that each triple really represents a path from SOURCE to SON of length DISTANCE, where the next-to-last hop on that path is FATHER. The structure LIST is initialized to contain the zero-length path from SOURCE to itself. At the first iteration, this path is removed from LIST, and SOURCE is placed

on the tree. Then the single-hop paths from SOURCE to each of its immediate neighbors are placed on LIST. At the second iteration, the shortest of these paths (i.e., the path to the closest immediate neighbor) is removed from LIST, and the closest neighbor of SOURCE (call it N) is placed on the tree as a son of SOURCE. Note that there cannot possibly be any shorter path to N, since any such path would have to have as its first hop a neighbor which is not as close to SOURCE as N itself. Therefore, the shortest path to N has been found. At this point, there is a possible path to each neighbor of N -- the two-hop path from SOURCE via N to N's neighbors. Each of these paths is placed on LIST. With further iterations, every path to each destination node is eventually encountered.

Note the way in which the paths are generated: at every iteration, the shortest path on the LIST is removed, and the LIST is augmented (if at all) only by paths which are one-hop extensions of that shortest path. It can be shown (see e.g., [1]) that all paths of length n will be encountered before any path of length > n is ever removed from LIST.

Naturally, since there are many possible paths to each destination N, and only one of those paths can be the first encountered. When a path to a node N is not the first encountered, special action must be taken.

Two cases exist. N is in the tree or N is on the LIST. If the node N is already in the tree, then some other path to N must have been previously encountered, placed on LIST, and removed from LIST. For that to be the case, the previously encountered path must be shorter than the one presently encountered. Hence the latter path need not be further considered, and should just be discarded and not placed on LIST. (See step 4a.)

Alternatively, the node N may not be in the tree yet, but a previously encountered path may already be on the LIST. If the path already on LIST is the shorter, it remains on LIST (step 4c). If not, it is removed from LIST and the lately encountered path is placed on LIST (step 4d).

When the LIST becomes empty, that means that all possible paths have been encountered, and the shortest path to each node found. At this point, the shortest path tree has been completed.

6.2.2 Incremental Algorithms

The manner in which incremental changes are accomplished is of prime importance because such changes must be handled rapidly in order for the algorithm to be viable. Re-calculation of the entire tree is probably too time-consuming to be performed whenever a change in the status of some line or node occurs, and therefore, algorithms to handle the possible network changes have

been developed. As will be shown in subsequent sections, a given incremental change to the network topology generally affects the routing to only a fraction of the nodes, causing only a minor change to a node's shortest path tree. Occasionally, however, a major portion of the tree must be restructured; thus it is important that when a change does affect the node its calculations can be done efficiently. Of course, there may be situations where it is more expedient to perform a comprehensive calculation than to do an incremental calculation.

The paragraphs below provide a qualitative description of the steps that have to be performed when each type of incremental change occurs. Then a single algorithm which consolidates all these steps with the basic algorithm is presented.

Line Changes

Assume that the shortest path tree for the source node prior to the change is known. First consider the case where the "length" of the line AB from node A to node B increases (e.g., the delay gets worse). Clearly, if the line is not in the tree, nothing need be done. If the line is in the tree, then the distances to B and to all nodes whose route from the source is via B increase. Thus, the nodes in the subtree whose root is B are candidates for changed routing. Conversely, routes to nodes not in this subtree will not be altered.

The first two steps for handling an increase in distance along a line are thus:

1) Identify nodes in B's subtree and update their distances from the source.

2) Try to find a shorter path to each subtree node S by routing S via those of its neighbors which are not in the subtree; if such a path is found, put node S on LIST. (More precisely, put the triple representing S on LIST.)

At the conclusion of these steps, LIST either will be empty or will contain some subtree nodes for which better (but not necessarily best) paths have been found. In order to find the best paths to the nodes on LIST as well as to the other subtree nodes, a slightly modified version of the SPF algorithm described in the previous section can be called. The modification that is necessary is step (4a), which skips nodes that are already on the tree. This procedure is correct when the tree is being generated from scratch, since then the algorithm ensures that once a node is placed on the tree, the shortest path to that node has been found. In the incremental case, however, the change in line or node status sometimes necessitates that a node be relocated. This modification is included in the consolidated algorithm given in the next section.

Now assume that the distance on the line from A to B decreases. If this line is in the tree for a given source node, then clearly paths to the elements of the subtree which has B as root will be unchanged because the subtree nodes were already at minimum distance, and hence the decreased line length will only shorten their distances from the source. Moreover, any node whose distance from the source is less than or equal to B's new distance from the source will not be re-positioned, since the node's path must reach B first in order to take advantage of the improved line. However, nodes which are not in the subtree and which are farther from the source than B may have a shorter distance via one of the subtree nodes.

The algorithm must thus first perform the following steps:

1) Identify the nodes in the subtree and update their distances from the source.

2) Try to find a shorter distance for each node K that is not in the subtree but is an immediate neighbor of a subtree node by routing K via those of its neighbors which are in the subtree; if such a path is found, put node K on LIST.

At the conclusion of these steps, LIST will contain some (possibly zero) subtree neighbor nodes that have been re-routed.

Neighbors of these nodes that are not in the subtree are candidates for improved routes also, and starting with the LIST generated in step 2 above, the modified version of the SPF algorithm can be used to restructure the rest of the tree.

If the line from A to B improved, but was not originally in the shortest path tree for a given node, then the algorithm must first see whether the node can take advantage of this improvement. Since the distance from the source to node A cannot be improved, the distance to B using the line AB will be equal to the distance to A plus the new distance along AB. If this updated distance is greater than or equal to the original distance from the source to node B, then the improved line does not help and no changes are made to the tree or to the routing table. If, on the other hand, the updated distance is less than the original distance, then the best route to B will now use AB. The first change to the shortest path tree is therefore to relocate B (and hence its subtree), attaching it to node A via line AB. Now the situation is identical to that of the previous paragraph in which the line from A to B was in the tree in the first place and its distance decreased.

Node Changes

First consider the case where node B goes down. This situation is easy to handle with a slight modification of the steps for line changes described above; namely, the nodes that must be relocated are those contained in B's subtree. This can be accomplished exactly as in the case for an increase in line length except that B is excluded from the subtree, since this node is now inoperative.

If, on the other hand, node B has been down but now comes up, the procedure is first to find the best route to B itself and then to find nodes that have improved distance by taking advantage of the fact that B is now operative. Specifically, the first step is to find the shortest distance to B by examining direct routes from each of B's neighbors. Node B is then put on LIST. Candidates for re-routing are all nodes whose distance from the source is greater than B's distance, and the basic SPF algorithm starting with LIST containing only B, can be used to restructure the tree.

6.2.3 Consolidated Routing Algorithm

The basic SPF algorithm and all of the incremental cases can be consolidated into the algorithm given below.

0. If no tree exists, put the source node on LIST, and go to step 7.

1. If the change was to the status of node B set DELTA infinite, and if B went down, then go to step 4, else go to step 3.

2. If the change was to line AB, then perform one of the following steps:

   a. If AB is in the tree, set DELTA equal to the change in distance along AB.

   b. If AB is not in the tree, set DELTA equal to the distance to node A plus the distance along AB minus the distance to B; if DELTA is greater than or equal to 0, done.

3. Identify node B as a member of the subtree.

4. Identify all of B's descendents (both first generation and succeeding generations) as members of the subtree.

5. Increase the distances of all subtree members by DELTA.

6. For each subtree node S perform one of the following steps:

   a. If DELTA is positive, try to find a shorter path to S via each of S's neighbors that is not in the subtree; if such an improved path is found, put the triple representing S on LIST.

b. If DELTA is negative, try to find a shorter path to each of S's non-subtree neighbors by attempting to route each neighbor via S; if such an improved path is found, put the triple for the neighbor node on LIST.

7. Search LIST for the triple T with the smallest DISTANCE.

8. Remove T from LIST.

9. Place SON(T) on the shortest path tree so that its father on the tree is FATHER(T). [Exception:
If SON(T) = SOURCE, place it in the tree as its root]

10. For each neighbor N of SON(T), do one of the following steps:

a. If N is already in the shortest path tree,

i) If its distance from SOURCE along the tree is less than or equal to DISTANCE(T) + LINE_LENGTH(SON(T), N), do nothing

ii) If its distance from SOURCE along the tree is greater than DISTANCE(T) + LINE_LENGTH(SON(T), N), remove N from the tree and place <N, SON(T), DISTANCE(T) + LINE_LENGTH(SON(T), N)> on LIST

b. If there is no triple T' on LIST such that SON(T') = N, then place the triple <N, SON(T), DISTANCE(T) + LINE_LENGTH (SON(T), N)> on LIST.

c. If there is already a triple T' on LIST such that SON(T') = N, and if DISTANCE(T') less than or equal to DISTANCE(T) + LINE_LENGTH (SON(T), N), do nothing.

d. If there is already a triple T' on LIST such that SON(T') = N, and if DISTANCE(T') > DISTANCE(T) + LINE_LENGTH (SON(T) N), then

   i) Remove T' from LIST

   ii) Place the triple < N, SON(T), DISTANCE(T) + LINE_LENGTH (SON(T), N) > on LIST

11. If LIST is non-empty, go to 7. Otherwise, the algorithm is finished.

6.3 Analysis and Data

When a given node receives an update message indicating that some line has gotten worse, the amount of time it takes to run the incremental SPF algorithm should be roughly proportional to the number of nodes in that line's subtree of the given node's shortest path tree. That is, it is roughly proportional to the number of nodes to which the delay has gotten worse. When a given node receives an update message indicating that some line has gotten better, the amount of time it takes to run the incremental SPF algorithm should be roughly proportional to the number of nodes in that line's subtree after the algorithm is run. That is, it is roughly proportional to the number of nodes to which the delay got better.

6.3.1 Average Subtree Size

One important measure of the efficiency of the SPF algorithm is its average running time. As indicated above, we expect that this is closely related to subtree size. Appendix 2 provides a simple derivation for a remarkable result:

-- In any tree, the average subtree size is equal to the average path length from the root to all nodes.

Figure 6-2 provides a summary of the proof of this statement for the ARPANET, in which the average path length is 5.5.

Average Path Length

$$h = \frac{1}{n-1} \sum_{i=1}^{d} i A_i$$

d = Depth

A_i = No. of Nodes at Depth i

Average Subtree Size

$$s = \frac{1}{n-1} \sum_{i=1}^{d} A_i \cdot \bar{B}_i$$

$\bar{B}_i$ = Average Subtree Size at Depth i

$$= \frac{1}{n-1} \sum_{i=1}^{d} \sum_{j=i}^{d} A_j$$

$$= h$$

In Regular Graph (Connectivity C)

$$s = h = d\left(\frac{(c-1)^d}{(c-1)^d - 1}\right) - \frac{1}{c-2}$$

Figure 6-2 Average Subtree Size Equals Average Path Length

Furthermore, the fraction of all the network lines appearing in a given shortest path tree is 1/c, where c is the number of lines per node. Thus, if we denote average path length as h, we have the useful result:

-- The expected subtree size of a given line in any node's shortest path tree is h/c.

Appendix 2 shows that this provides an upper bound on the running time of the SPF algorithm, and gives some numerical values for h/c as a function of network size.

Figure 6-3 shows the significance of these results for the ARPANET. (The running time of 1 to 2 milliseconds is discussed in more detail in section 6.3.3 below.)

6.3.2 Distribution of Subtree Sizes

In order to determine the distribution of subtree sizes for all lines in the ARPANET, we used the comprehensive SPF algorithm to construct shortest path trees for all nodes in a network. Then we simply counted the subtree size of each line. We tested many sets of line lengths and found no significant differences in the average subtree size. For non-congested networks, the average subtree size is about 2, and the median subtree size is about 2. A surprising 1/3 of all nodes are leaves (have no descendants). A total of 85% of all lines have subtrees which

	In General	ARPANET
# Nodes	N	62
# Lines	L	75
Connectivity	$c = 2 \cdot L/N$	2.5
Average Path Length	h	5.5
Average Subtree Size For Lines in Tree	s = h	5.5
% of Lines in Tree	1/c	40 %
Average Subtree Size Overall	h/c	2.18
Expected Running Time For SPF	$k \cdot \frac{h}{c}$	1-2 ms

Figure 6-3 Performance of SPF Algorithm

are smaller than 11 nodes. A total of 95% of all lines have subtrees which are smaller than 21 nodes. These facts are consistent with the theoretical predictions, indicating that very few of the lines in the network ever have very large subtrees.

We also modeled a badly congested, or damaged, network and found the average subtree size to be about 5, and the median to be about 9. The incremental SPF algorithms will, on the average, take longer to run in a congested or damaged network than in a non-congested network.

6.3.3 Measured Performance of SPF

In order to get an idea of the absolute amount of space and time it takes to run the SPF algorithm, we programmed SPF in FORTRAN (actually RATFOR, a structured programming version) on TENEX. Then we coded the algorithm in assembly language for the 316 IMP and Pluribus (1 processor) IMP, using the TENEX version as a model. We were pleased to find all three versions required approximately the same amount of storage, as shown in Figure 6-4.

To determine running times, we randomly assigned each line in the ARPANET a length between 1 and 20. We ran the comprehensive SPF algorithm to initialize the data structure in each node. Then we used a random number generator to generate, in sequence, 50 routing updates. That is, we picked 50 lines at

random and successively gave each a new random length. Every time we changed the length of a line, we changed it by at least 15%. Also, some lines were brought down by being assigned a length which represented infinity. Each time we did this, we ran the incremental SPF algorithm in all the nodes. We obtained the following two empirical results for TENEX (surprisingly similar results hold for the 316 and Pluribus implementation):

a. The average time per IMP to run the incremental SPF algorithm was 2.2 milliseconds.

b. The average time per subtree element to run the incremental SPF algorithm was 1.1 milliseconds. That is, if, for a given node, a particular update affected the distance to N destination nodes, the algorithm would take 1.1N milliseconds, on the average, to run in that node. (The standard deviation was 0.46.)

Since we know that average subtree size for a non-congested network is about 2, these two results are in agreement. Note that this is only an average figure. Actual times vary from under 1 millisecond to 40 milliseconds. We also ran the algorithm 50 times on a similar network, except that certain lines were given "length" 80, to simulate a congested or damaged network. The average time per IMP to run the incremental algorithm increased slightly, to 2.4 ms.

	TENEX 36-Bit Words Fortran	PLURIBUS 16-Bit Words Assembly	H 316 16-Bit Words Assembly
Size (words)			
Module: MAIN	29	27	23
RTALL	42	42	37
RTINC	70	63	72
RETREE	128	110	133
SEARCH	67	50	66
ROUTER	78	85	108
Total	414	377	439
Running Time (ms.)			
Complete Tree			
Min	47	34	47
Max	59	39	53
Ave	52	36	50
Updates (50 Trials)			
Min/Trial	1	0.6	1.1
Max/Trial	4.4	2.7	4.2
Ave	2.2	1.3	2.2

Figure 6-4 Results from Programming SPF

One valuable lesson we learned in this investigation is that we can use the storage and timing results from one implementation of an IMP algorithm on TENEX, the 316, or the Pluribus, to predict quite closely the performance of that algorithm on the other machines.

6.4 Updating Policies

In previous sections we noted some of the deleterious effects of the present ARPANET periodic routing policy--in particular, the speed of adaption to network changes, the excessive average line bandwidth used by routing messages, and the delay induced in data traffic. Section 5.1 describes an alternative technique, event-driven updating, in the context of the ARPANET routing algorithm. The advantages of event-driven updating clearly apply also to the SPF algorithm, and here we discuss some of the issues involved in developing such an updating policy for a routing algorithm like SPF, which requires an identical data base at all the nodes. The updating technique we design must meet the following criteria:

	Normal Operations	IMP failure or partition	IMP recovery or partition end
Efficiency	Low CPU and line overhead	Fast notification at low overhead	
Reliability	Sequencing of multiple updates	No loss of updates	Complete information made available

Five questions, about update data, addressing and routing, error control, topology changes, and program implementation are considered below. Some questions are answered; others remain open for now; the conclusions are summarized at the end of the section.

6.4.1 Data Contained in the Update Message

a. One line vs. all lines at that IMP?

Sending information about all the lines at the IMP has several advantages:

-- more efficiency under heavy loading/many changes (less than a factor of c more efficiency, where c = number of lines/node).

-- less ambiguity about the sequencing of multiple updates (when a node receives an update about one line at an IMP, it gets synchronized information about the status of the other lines).

-- fewer serial numbers required: one per IMP, not one per line (note: the size of this number should be determined by some analysis; we have not yet done so).

-- simplicity of performing periodic re-broadcasts if desirable.

-- all lines can be updated somewhat more frequently, so accurate "event" detection and event-driven updating is less crucial.

-- the most recent update contains all necessary information; an ack system which checks only the last number received is adequate.

Sending only one line's data permits a restriction of no more than x updates/time period/line, but that seems not to be an important advantage.

b. One IMP's data or several IMPs'?

Certainly, the ability to combine several update blocks into a single network message is an important efficiency measure. This would be useful at the end of a network partition, when several IMPs need to report information to each other.

c. How should lines be identified? How should topological changes be updated?

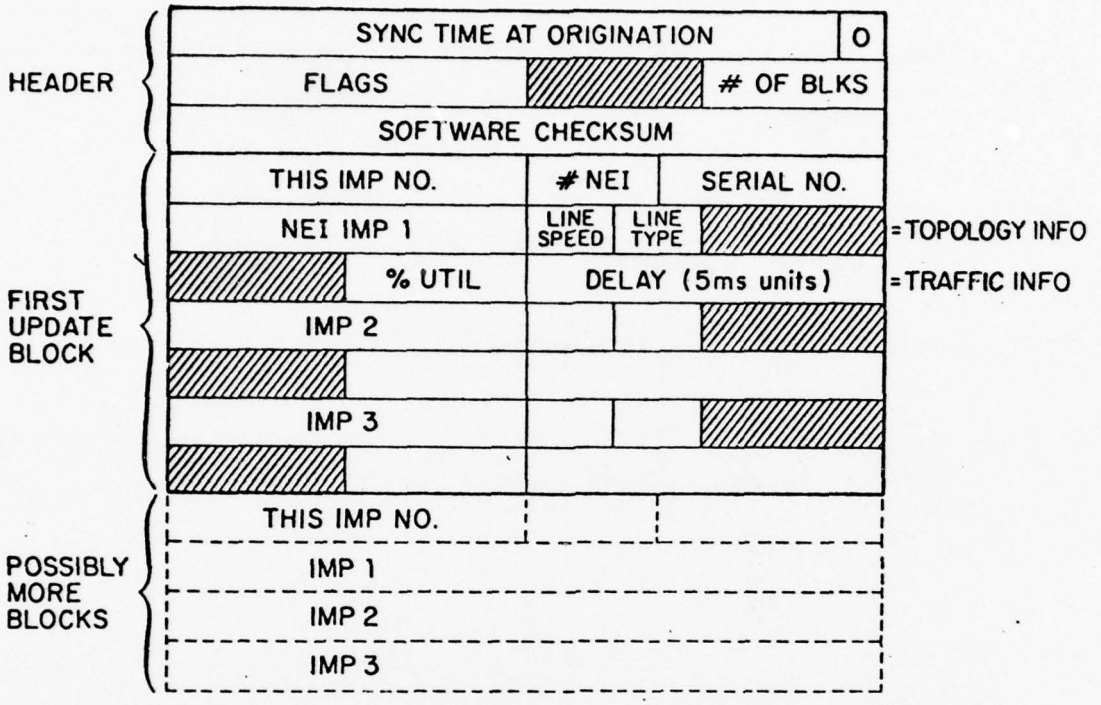
It seems most reliable, general, and certainly simple enough to identify each line by the numbers of the IMPs at each end rather than by any artificial numbering. This permits any topological change (e.g., modem wire) to be effected easily.

d. What information is needed for analysis and debugging?

-- SYNC time at origination, to tell how long an update takes

-- Serial numbers on each update block, to detect missed updates

e. What should the format be?



Length(bits): Hardware framing 72
Software header 48
Each block 16 + 32* #NEIs

Some example updates
1 Update, 2-line node = 200 bits 4ms @ 50 Kbps
1 Update, 3-line node = 232 bits 4.6 ms @ 50 Kbps
9 Updates, 3-line nodes = 1128 bits 22.6 ms @ 50 Kbps

Fig. 6-5 Routing Update Format

Figure 6-5 shows a suggested format. Note that there is no address (no to: field) in the update; this question is considered in the next section.

6.4.2 Addressing and Routing Updates

There are two general types of approaches:

a. "Broadcasting", in which the source addresses the update explicitly to all nodes, and routes it to each node along the best path. (This is discussed in ARS #23.) Such a scheme requires N-1 packet hops for the broadcast, which is optimal.

b. "Flooding", in which each node sends each new update (one it has not seen before: the serial number is larger than the last one received about that IMP) on all its lines except the line on which the update was received. This requires L-N+1 packet hops (where L is the number of lines in the net counting each direction), since an update will flow on all lines except "backwards" on the N-1 lines of the broadcast tree from the source. If we define L=cN, where c=average connectivity, then this number of updates is cN- (N-1) = (c-1)N +1.

What can we say about these two methods, A and B? First, let's consider efficiency. The most important consideration in efficiency is line bandwidth; CPU bandwidth requirements can be shown to be very small.

Method A: The message is b+N bits long, where b is the number of bits in the body and N is the N-bit address. Thus the

total number of bits on all lines is (b+N)(N-1). Therefore, the total bit rate per line if each node updates every t seconds is

$$\frac{N(b+N)(N-1)}{cNt} = \frac{(b+N)(N-1)}{ct}$$

Method B: The message length is only b bits, thus the total number of bits on all lines is b((c-1)N+1). The total bit rate per line is

$$\frac{bN((c-1)N+1)}{cNt} = \frac{b((c-1)N+1)}{ct}$$

Method A is quadratic in N, which means for large enough N it will become more expensive than Method B. The crossover point comes when

$$(b+N)(N-1) > b((c-1)N+1)$$

The exact solution has a messy form; it can be approximated as

$$N > b(c-2)$$

That is, when N > b(c-2), Method B is more efficient. This is illustrated in Figure 6-6 below for b=200 bits, t=100 seconds. For c=2.5 (ARPANET), crossover comes near 200*.5=100 nodes. For c=4, crossover is not until 400 nodes. Note, however, that for c=2.5 both methods A and B require less than 100 bps (.2% of 50 Kbs) to update 80 nodes at a rate of once every 100 seconds. The

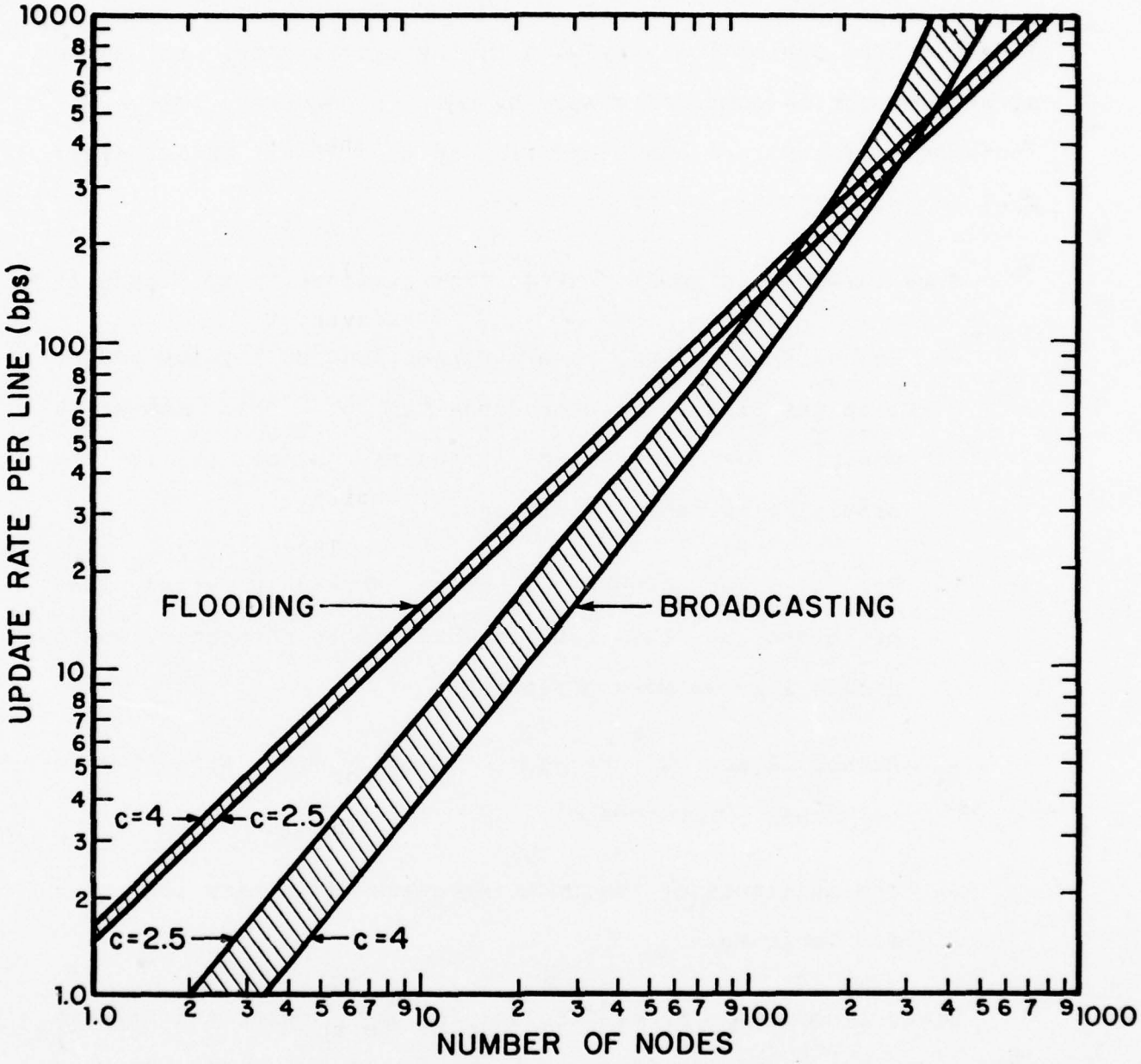


Fig. 6-6 Routing Overhead per Line (assuming 1 update/node/100 seconds)

line overhead scales linearly with t, the update rate, so other strategies can be compared simply by relabelling the y axis. For instance, if t=10 sec, then updating in the ARPANET (N-62, c=25) would require 750 bps (1.5% of 50 kbs).

Some interesting points emerge from examination of Figure 2:

-- Method B (flooding) is a straight line on log-log paper, with practically no dependence on c. This makes it useful for long-range planning, since it is not sensitive to network topology.

-- For a given number of nodes Method B grows less efficient as the net is more highly connected, while Method A grows more efficient.

-- Methods A and B are quite similar for ARPANET-size networks (50-200 nodes).

-- The magnitude of the updating overhead is very low, even for large nets.

Since flooding is more efficient for large nets, and is very efficient in absolute terms for small nets, it is the best overall choice for efficiency. For the ARPANET, the two methods have nearly identical efficiency. The choice between them should be made on other grounds.

A second important advantage of flooding is that the IMP sends the same message on all its lines, as opposed to creating separate messages with different bit-vector addresses on the different lines. This may make it considerably simpler to program the update mechanism, since there is no problem of reserving buffers or dealing with the situation in which the IMP has no more buffers for update copies.

A final consideration which favors flooding is that it does not depend on the correct operation of the routing algorithm. This makes it a safer, more reliable system than broadcast. This also covers the special case of sending updates on dead lines, which helps to speed the update process, and avoid various unlikely error cases (e.g., one line at an IMP comes up at the same time its other line goes down, so it misses an update).

6.4.3 Reliable Transmission of Updates

a. Should updates be acknowledged? How?

At first glance, it seems essential to ack updates to make sure they get through, and this is useful for flooding:

> If updates are acked at each hop, then with flooding the update will be received at all nodes which have a path to the source at the time of the update.

On the other hand, with broadcasting using an n-bit vector, transmission is not reliable. Two examples are (1) a node which receives an update, acks it, and then fails, and (2) a section of the network which is partitioned from the rest while an update is flowing through it destined for the main body of the net. In each case, an update will be lost.

> Under broadcasting, acking updates at each hop is not sufficient to ensure reliable updating of all nodes which have a path to the source at the time of the update.

If we decide to use a positive ack/retransmission system for routing updates, then we have to build the right data and control structures in the IMP. Some possibilities are:

Method	Problems
1. Use regular modem logical channels and acks	Need different channels for transmission on different lines. What to do if no channels are available.
2. Invent a new set of logical channels for routing, common to all lines.	Adds complexity to the IMP (packets on multiple queues, etc.)
3. Send all acks periodically rather than one at a time in separate messages.	Slower reaction to a lost update than usual ack system.

One last possibility, though not an ack system, is:

4. Send the update once only, and rely on a periodic retransmission to ensure it gets through.	Even slower error recovery, though very reliable in the long run.

Method 1 is not very workable, since there are problems associated with copying the update into several buffers for the different outputs, setting up channels and queues in a non-disruptive way, and dealing with the case of no available channels.

For Method 2, the ack would look very similar to today's packet acknowledgments, though there might be a need for more than 8 ack channels if we want to make very sure there is no blocking for lack of channels. The expected number of updates per ack would be very small, since there are very few updates/line/second with either broadcasting or flooding:

broadcasting: $\frac{N-1}{ct}$; for ARPANET = $\frac{25}{t}$ = 25 times routing update rate

flooding: $\frac{(c-1)N+1}{ct}$; for ARPANET = $\frac{35}{t}$ = 35 times routing update rate

Thus, if routing updates were generated once a minute on average by each node, then an update message would flow over each line every two seconds on average.

For Method 3, we could use the periodic Hello/IHY message exchange to carry a one-bit odd/even bit for each node. This would require an additional N bits per Hello/IHY message, rather than a separate ack message for each update. A drawback of this scheme is that the sender is "blocked" from sending another update about some IMP until the previous one is acked.

We now compare these two possibilities in terms of the extra line overhead they require. In both cases, we will assume a 136-bit Hello/IHY message is sent every 640 ms, contributing 212 bps of overhead, and we are concerned only with additional overhead beyond this:

	Method 2 Separate Acks	Method 3 All Acks in Hello
broadcasting	$\frac{136(N-1)}{ct}$ bps	$\frac{N}{.64}$ bps
flooding	$\frac{136((c-1)N+1)}{ct}$ bps	

This assumes separate acks are 136 bits long. We can solve various equalities to show the following statements are true for all values of N (all network sizes):

-- Separate acks with flooding use 50% more bandwidth than separate acks with broadcasting.

-- All acks in the Hello are better than separate acks if t, the routing update period, is small:

broadcasting: if $t<34$ secs, Method 3 is better.
flooding: if $t<52$ secs, Method 3 is better.

-- Any ack method will contribute significantly to line overhead, as much as doubling the routing overhead.

The amount of line overhead used by the two ack methods is shown in Figure 6-7, for both broadcasting and flooding in the case of separate acks, for t=10 secs, and t=100 seconds. (t is the average time between routing updates generated by each node.)

The total overhead associated with routing is the sum of the routing update, acknowledgment, and Hello/IHY overheads. For the cases under consideration, we have

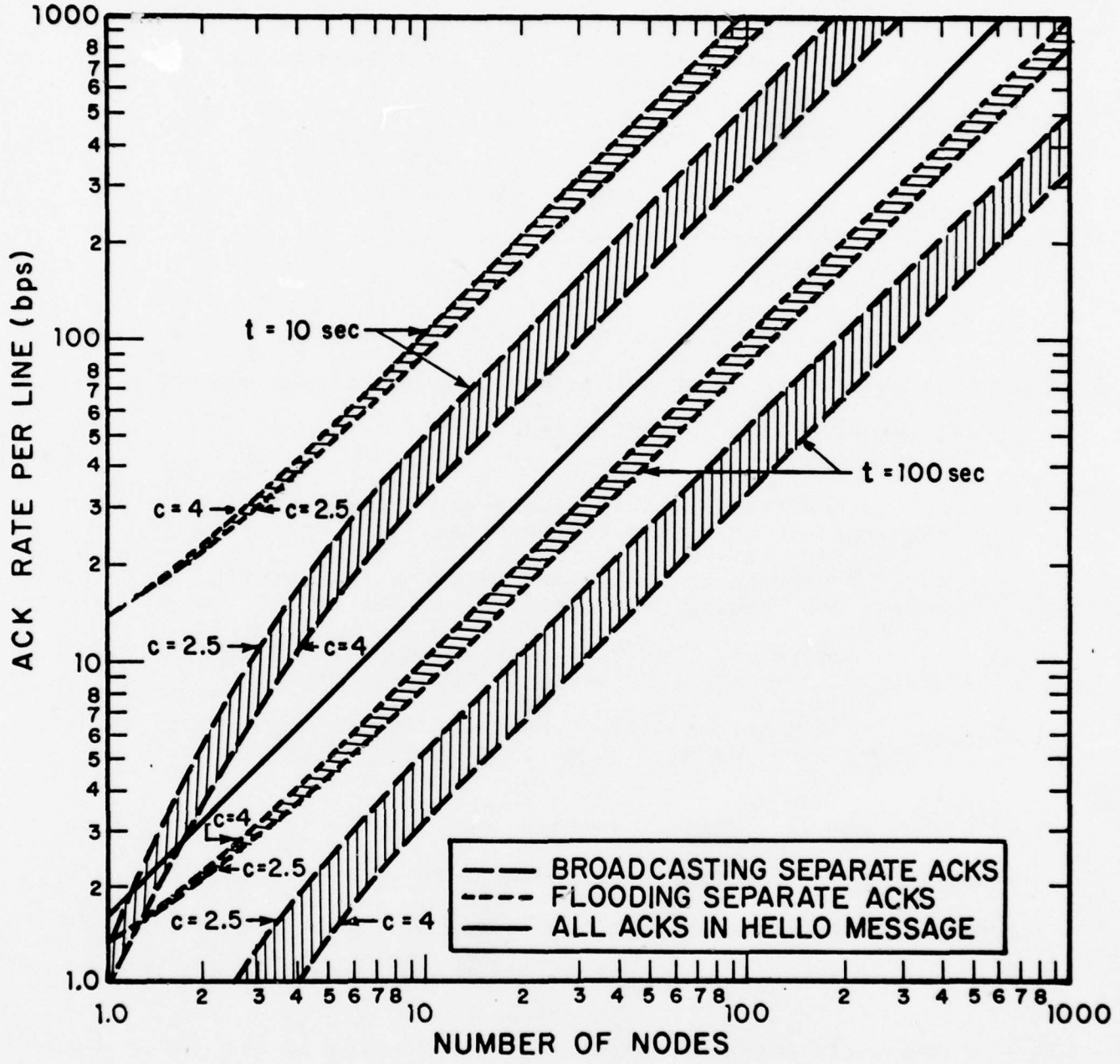


Figure 6-7 Acknowledgment Overhead Per Line

	Routing	Ack	Hello	Total
Broadcasting:				
Separate Ack	$\frac{(200+N)(N-1)}{ct}$	$\frac{136(N-1)}{ct}$	$\frac{136}{.64}$	$\frac{(336+N)(N-1)}{ct} + \frac{136}{.64}$
Ack in Hello	$\frac{(200+N)(N-1)}{ct}$	$\frac{N}{.64}$	$\frac{136}{.64}$	$\frac{(200+N)(N-1)}{ct} + \frac{136+N}{.64}$
Flooding:				
Separate Ack	$\frac{200((c-1)N+1)}{ct}$	$\frac{136((c-1)N+1)}{ct}$	$\frac{136}{.64}$	$\frac{336((c-1)N+1)}{ct} + \frac{136}{.64}$
Ack in Hello	$\frac{200((c-1)N+1)}{ct}$	$\frac{N}{.64}$	$\frac{136}{.64}$	$\frac{200((c-1)N+1)}{ct} + \frac{136+N}{.64}$

These four cases are compared in Figure 6-8,

for t=100 sec
c=2.5

As an additional precaution, especially during test and installation, we can use Method 4 to periodically re-flood the net with the routing information from each source. For instance, we could set the periodic rate at 1/100 seconds, which would add only 80 bps to each network line. (This can be accomplished by a method like we use for cumstats to stagger the transmissions by IMP number over the whole time interval. The interval can be 105 seconds=25.6 ms*4096.)

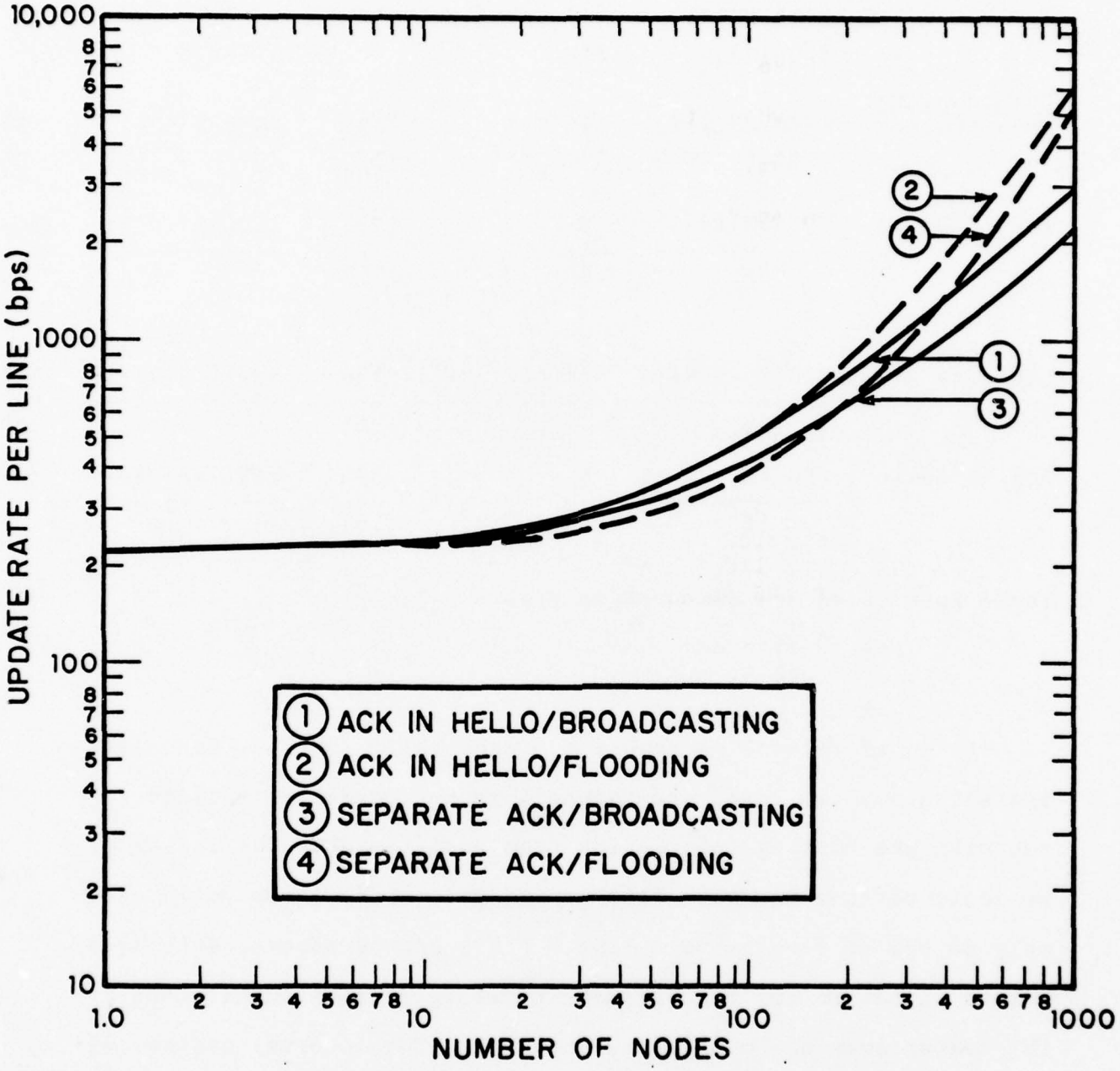


Figure 6-8 Total Overhead Per Line

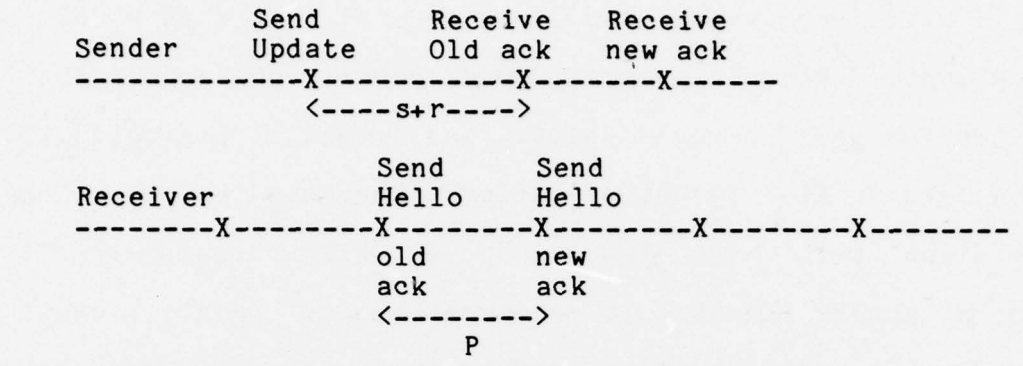
With any ack method, there is the chance of further foul-ups. With a separate ack scheme, the sender must keep a timer for each un-acked update, and resend it periodically. With the acks carried in Hello messages, there is the chance that the receiver will have just sent a Hello containing the old serial number bit when it receives the new update, causing the sender to retransmit the update unnecessarily. The probability of the update and an "old" ack crossing in mid-flight is

$$\frac{s+r}{p}$$

s = time to send update
r = time to send Hello/ack
p = period for sending Hellos (0.64 sec)

For typical ARPANET land lines s and r will be equal and small:

queueing delay	10 ms
transmission delay	5 ms
speed of light delay	5 ms
processing latency	5 ms
TOTAL	25 ms for s or r



Thus $\frac{s+r}{P} = \frac{50 \text{ ms}}{640 \text{ ms}} = 8\%$ spurious retransmissions.

Alternatively, the IMP could keep a clock for each destination which would ignore any Hello/acks within the last x ticks (e.g., a two-bit counter of 25 ms ticks that is set to 11 at update time, and is then run down to 00 before any retransmissions are attempted). This would be necessary on satellite lines, where the probability of retransmission without the timer becomes close to 1. There the timer must be longer, e.g., 600 ms.

6.4.4 Updates about Topology Changes

a. When a single line comes up or goes down, it is reported by flooding the net with the corresponding update.

b. When a single node goes down, it is reported as multiple lines going down. It is too difficult to determine if a

node has gone down in any "direct" manner. When a node is detected to be unreachable, the table entry for that node should be marked with a "dead" bit indicating that its line delay values are invalid. When it comes up again, the bit can be cleared.

c. It is unnecessary to explicitly report a node coming up; the fact that its lines come up is enough. On the other hand, we might prefer to wait for an update from that node to clear the "dead" bit.

d. When a node comes up, or when it returns from a partitioned state (in which it was isolated from several other network nodes -- typically when its line(s) to the network were down) it must get a complete update of all network routing information, since an indeterminate number of updates have taken place. Two possibilities exist:

(1) The two adjacent nodes which were isolated from each other send each other their entire routing tables (delay information on all lines).

(2) The two adjacent nodes exchange all table entries for which the dead bit is off.

The first is somewhat simpler to program, but uses more line bandwidth than the second. If the table is garbage-collected or compacted to remove entries for unreachable nodes, then the two methods are identical. If compaction is not too difficult, it is probably the best method.

e. When a node receives such an update, possibly containing information on many nodes previously unreachable, it treats it like a normal single-node update, and sends it to its neighbor nodes by the flooding method. This works well for both sides when an IMP that was down comes up; it gets all the tables from its neighbor, and the rest of the net gets its own single table entry.

f. Several messages may be required to send all the routing information to a node or nodes which were previously isolated. (About 15 node tables can be packaged in each packet.) Although it will not lead to optimal routing during the transition, it seems simplest for each node to process all the updates as they arrive, and to start sending traffic on those paths before receiving all the other updates.

g. There is no need for an IMP Coming Up state or timer, since routing should flow as quickly as data packets.

h. There is no need for an IMP Going Down state or timer, since reachability is determined explicitly.

6.4.5 Program Modules for Updating

a. Receive update:

```
                                                        Level
For each IMP-block in update:                           Modem In
   If serial no. > present serial no:
      Copy data into table
      Mark which line(s) changed
      Update one-bit serial no/ack

If some serial no. is new:                              Modem In
   Queue update packet for output
   on all other lines

If some line changed:                                   Fast Timeout
   For each line that changed:
      Run incremental SPF
```

b. Send update:

```
If update packet queued:                                Modem Out
   Remove from queue, transmit
   Set timer to ignore acks for x secs?
   After transmission, call FLUSH to free packet
```

c. Generate update:

```
If any line has gone down, come up, or changed          Slow Timeout
delay significantly, or if 100 seconds elapsed
since last update
   Increment serial number by 1
   Copy all line data into update packet
   and queue for output
```

d. Receive acks in Hello:

```
For each IMP:                                              Modem In
   If received ack not equal to present serial no:           or TASK
      copy IMP's data into update buffer
      If buffer is full, queue for output
      on this line
If update buffer not empty:
   Queue for output on this line
```

e. Send acks in Hello:

```
Copy table of low-order bits of serial nos.                Modem Out
into Hello
```

Based upon the discussion above, we can state the following conclusions about event-driven updating policies for the SPF algorithm:

-- Flooding seems preferable to broadcasting.

-- Putting acks in Hellos seems a little better than separate acks, but there is the problem of what to do if a new update arrives before the old one is acked.

-- We should study the behavior of the algorithm under changing network topology, to determine how fast it works, how big the serial number should be, etc.

6.5 Comparison of SPF to Current ARPANET Routing

The SPF algorithm is similar to the current ARPANET routing algorithm in several ways. Both are single-path algorithms, and hence can never produce optimal routing. Also, the SPF algorithm can generate the same sort of routing table as currently exists. Therefore it would require no changes to the forwarding procedure.

The most fundamental difference between the two algorithms has to do with the fact that SPF is line-based, whereas ARPANET routing is path-based. That is, the ARPANET routing algorithm knows nothing of the topology of the network, and it does not know the state of any lines other than the lines which are local to the IMP the algorithm is running in. All the current algorithm knows is the aggregate delay along various paths it considers (it knows only the next hop on each path, and hence cannot distinguish among paths for which the next hop is the same.) SPF, on the other hand, must know the topology, as well as the state of each line in the network. This has an obvious disadvantage -- SPF needs a larger data base, hence more memory. However, SPF has many advantages:

1. Reachability. The ARPANET algorithm has to do everything twice, once to find the least delay paths (for

routing) and once to find the min-hop paths (for reachability considerations). Even so, the ARPANET is slow to detect unreachable nodes. When a node crashes, no other node can possibly declare it unreachable until a period of ten seconds elapses. In ten seconds, a lot of traffic for the unreachable node can build up on network queues, interfering with other traffic flows and causing network disturbances. This reachability problem is a feature of any path-based routing algorithm. When one or more lines go down, the IMPs can determine that they have lost their paths to several destinations, but they have no way of determining the cause of their losing these paths. Rather, the IMPs operate on the assumption that if they lose a path to an IMP which is not unreachable, a new path will appear within some maximum period of time (chosen in the ARPANET as ten seconds). Therefore, if no new path appears in that time, they assume the IMP is unreachable.

The SPF algorithm, on the other hand, makes it easy to determine reachability, and without the use of any heuristics. With the SPF algorithm, the IMPs must know the state of each line in the network. It follows that they know of each IMP whether it is reachable or not. This information is available as part of the regular routing algorithm, and would be available in much less than ten seconds.

2. Loops. Unless a routing algorithm ensures that all IMPs run their routing calculations off the same data base, it is possible that loops will form. That is not to say that the SPF algorithm produces loops within a given node's routing table -- the SPF algorithm is guaranteed to produce a loop-free shortest path tree in every node. However, if different nodes disagree about the state of some line, they may well produce conflicting trees, thereby causing packets to loop. However, the fact that an algorithm permits transient loops is only a very minor point against it. A loop is just a certain sort of sub-optimal traffic flow, and is inherently no more harmful than any other sort of sub-optimal traffic flow. Loops present a special problem only if they last for unusually long periods of time, or if they slow the convergence of the algorithm.

A) The ARPANET routing algorithm with hold-down permits loops to form and then persist forever, effectively making certain nodes spuriously unreachable from certain areas of the network. To prevent this, a special check has been added to the network to detect these loops and break them. The SPF algorithm, however, would never permit loops to last for more than an extremely short period. Updates on line status could be sent around the network very quickly, so that there could be only a very short period of time when nodes disagree about the status of

a particular line. Hence, loops should be short-lived, and not particularly harmful. (Of course, this assumes that line status does not oscillate rapidly. Hysteresis in the line protocol prevents rapid oscillation between the up and down states. And the fact that the updates will be average values over some time interval with bias factor as necessary should prevent successive updates from oscillating.)

B) The convergence of the ARPANET routing algorithm is slowed down considerably when loops form. The reason for this is that the presence of loops contaminates the update procedure. Once a loop forms, many IMPs will be generating routing update messages containing false information. These in turn cause other IMPs to generate updates with false information, and the algorithm will not stabilize until all the false information is purged. With the SPF algorithm however, the presence of loops would have no effect on the conveyance of the algorithm. Since updates contain information about lines, rather than paths, the presence of loops could not contaminate the update process at all. Therefore, the loops would not be very harmful.

3. Amount of Work. The ARPANET routing algorithm runs at a particular frequency whether it has anything to do or not. Further, the amount of time it takes to run is completely independent of whether or not any routing changes need to be

made. The SPF algorithm, on the other hand, uses event-driven updates, so that it runs only when there is some work for it to do. Further, the amount of time it takes to run is directly proportional to the amount of work it has to do. If it is necessary to make a lot of routing changes, the SPF algorithm will take longer to run than the current ARPANET algorithms. But in the much more likely case that there are only a few routing changes to make, the SPF algorithm runs very quickly.

We note in closing that the ARPANET algorithm could be modified to calculate reachability based on delay, not to use hops, and to do event-driven updating, as described in Section 5. However, the SPF algorithm benefits more from these techniques than the ARPANET algorithm does.

6.6 Conclusions

The principal conclusions from our initial investigation of SPF are:

1. A shortest path algorithm which runs independently in each IMP is a practical alternative for ARPANET routing.

2. An incremental version of such an algorithm has very attractive characteristics, permitting efficient operation while facilitating rapid response to network changes.

3. The time required to generate the shortest path tree from scratch is not excessive (approximately 30-50 msec).

4. The average time required to update the tree for a network change is small (1-2 msec). This time is approximately proportional to h/c, where h is average path length and c is the average number of lines per node.

5. The worst-case time to update the tree for a single change is less than the time required to re-generate the tree.

6. The memory requirements for the algorithm and for the associated data structure are roughly 400 words and 280 words, respectively.

7. Programming SPF on TENEX first, and then re-coding for the 316 and Pluribus was a useful approach. Furthermore, running times and memory requirements for SPF showed close agreement among all 3 machines, suggesting that future results from one implementation can be used to predict performance on the others.

8. Efficient and reliable updating methods can be implemented for transmitting data about changes on any line to all nodes.

9. SPF has significant advantages over the present ARPANET algorithm, and further work should be carried out towards its eventual installation in the ARPANET.

7. MESSAGE ADDRESSING MODES

This section discusses the use of enhanced modes of message addressing in the ARPANET. The basic mode of operation of many computer networks today is that the subscriber presents a message to the network with a physical address corresponding to the destination subscriber's number. Three other possibilities have been considered for the ARPANET: logical addressing, in which one of several physical addresses is denoted by a single logical address, multi-destination addressing, in which a list of physical and logical addresses is provided, and group addressing, in which a single address denotes a group of logical and physical addresses.

Logical addressing is closely related to the issues of multiple homing of subscribers to network ports and the use of one network port for the connection of several distinct subscribers. In other words, a general network addressing structure should permit many physical addresses to correspond to a single logical address and one physical address to correspond to many logical addresses. The translation of logical addresses to physical addresses is an important problem and can be handled differently in virtual circuit networks and in datagram networks. In a virtual circuit network the address can be translated once per connection at the source, permitting all packets in a given

virtual circuit to flow to a particular physical address. On the other hand, in a datagram network the address of each message can be translated separately at the source node or at each intermediate node in the network. Packets can flow to any physical address.

The problem of failure recovery in a logical addressing and multi-homing environment is a complicated one. Failure recovery procedures can be specified in which all possible network status information concerning the virtual connection (or datagram flow) is communicated to the subscriber. However, there are certain to be situations in which the network cannot provide a subscriber with complete information concerning the extent of the failure.

Multi-destination addressing and group addressing is most easily accomplished for datagram networks since (1) it is not necessary to set up multiple connections for a particular transmission, and (2) complicated error control and flow control strategies needed with multiple message numbers, acknowledgments and allocations do not flow over the same multi-destination circuit. For a datagram with many addresses the problem is simply to route the datagram correctly and efficiently to the destinations.

The efficiency of a multi-destination or group addressing system depends critically on the routing algorithm used. One useful metric for determining the efficiency of a multi-destination system is the number of packet hops required to transmit a given packet to all the destinations. A routing algorithm which minimizes the number of packet hops needed for multi-destination transmissions can be constructed on the basis of a standard minimum hop routing algorithm for single destination packets. In addition to this algorithm, it is necessary to provide a multi-destination address in the header of the packet. When the packet arrives at an intermediate node, the node simply creates as many copies of the packet as there are different routes in the minimum hop routing table for the different destinations in the header. Each time multiple copies of a packet are created at a node, each copy is assigned the appropriate subset of the destinations for which that path is the minimum hop path. In this way a broadcast of a given packet to all n-1 other nodes in the network can be accomplished with only n-1 packet hops, which is optimal. (Note that other single-path routing algorithms such as a minimum delay routing procedure are also optimal for a broadcast to all other nodes, though not for a message addressed to fewer subscribers.)

The routing strategy we propose works as follows:

a. The address of a packet is an n-bit vector with bit I equal to 1 indicating that the packet should be sent to IMP I.

b. The routing table in each IMP can be represented as 1 n-bit vector per line in which a one-bit on line L for destination I indicates that line L is the best route to IMP I.

c. The routing decision can be carried out by each IMP receiving a message with a particular address subtracting its own address bit from the n-bit address vector and then performing a calculation to determine the addresses for the one or more packets which are to be sent out to the IMP's other lines. For each of the other lines L the new address equals the AND of the incoming address and the n-bit route vector for that line. A packet should be sent out on line L if the resulting address is not all zero.

We next analyze the performance improvement gained by multi-address messages, measured in packet-hops: the number of hops traversed by each packet summed over all packets transmitted. The following definitions will be useful:

n	Number of nodes in net
a	Number of addresses
p(a)	Number of packet hops
h	Average path length
c	Average node connectivity
d	"Depth" of SPF tree (see ARS #23)

For separately addressed packets, it requires h*a packet-hops on an average to transmit a packets. For multi-address, p(a) is a more complicated function:

	Chain of nodes	Ring of nodes	Fully connected net	General net
p(1)	$\frac{n+1}{3}$	$\frac{n+1}{4}$ (n odd)	1	$h = d - \frac{1}{c-2}$
p(2)		$2\,\frac{n+1}{4} - \frac{(n-3)(n+1)}{12(n-2)}$	2	2h -?
p(n-2)	$n-2 + \frac{n-2}{n}$	$n-2 + \frac{n-3}{n-1}$	n-2	n-2 +?
p(n-1)	n-1	n-1	n-1	n-1

A little thought shows that $p(a) \leq h*a$ for all a, and
$p(a) \geq a$ for all a.

Furthermore, $p(a+b) \leq p(a) + p(b)$; p(a) is concave downward.

Figures 7-1 and 7-2 below show some investigations we made to determine the behavior of p(a) for the ARPANET. The best fit we have for p(a) is approximately:

$$p(a) = ha - (h-1)a*\log_{n-1}(a)$$

That is, the percentage improvement of multi-address over separate addresses is given (approximately) by

$$(h-1)a*\log_{n-1}(a)$$

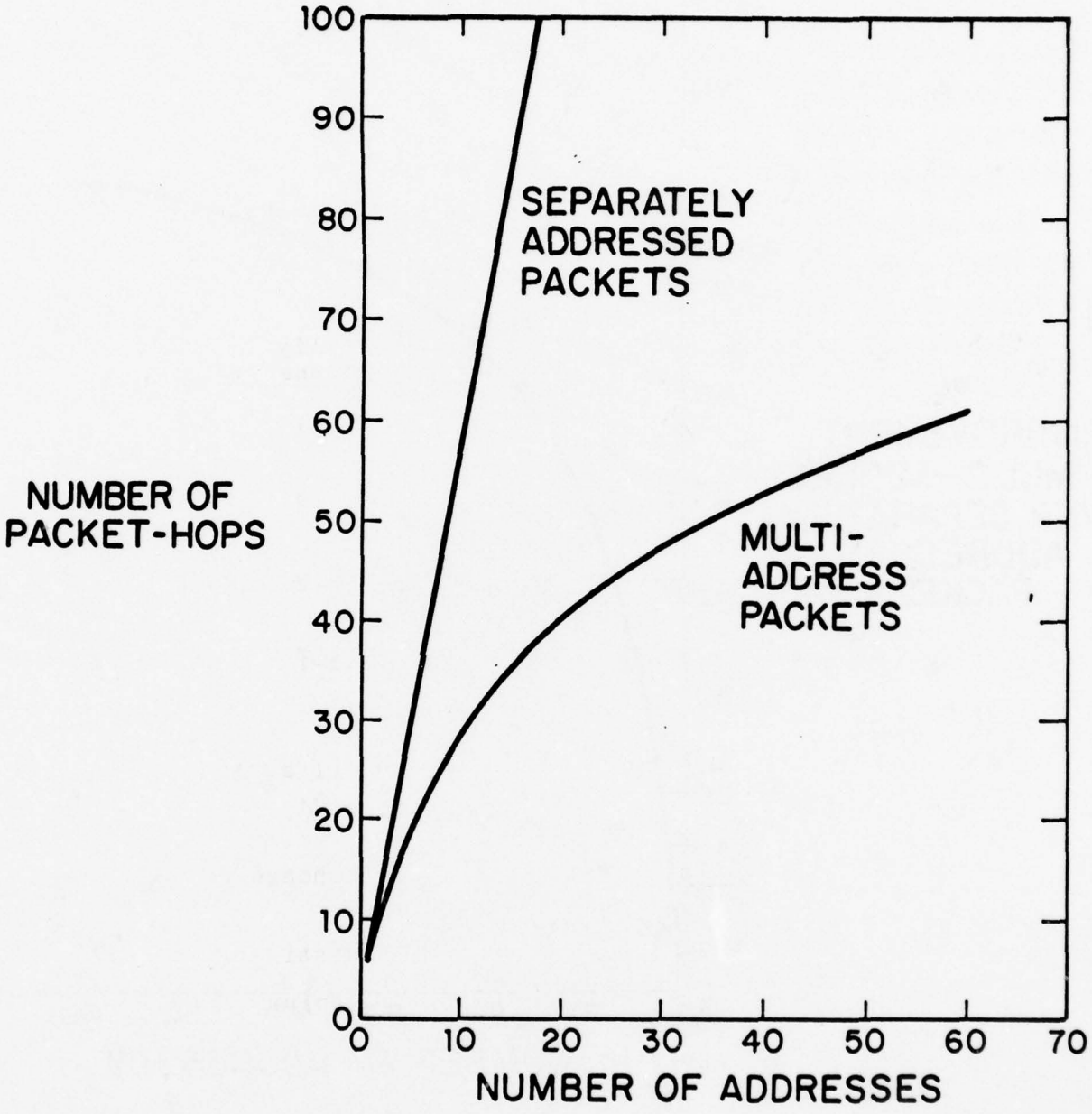


Figure 7-1 Multi-Address Packets: Number of Packet Hops

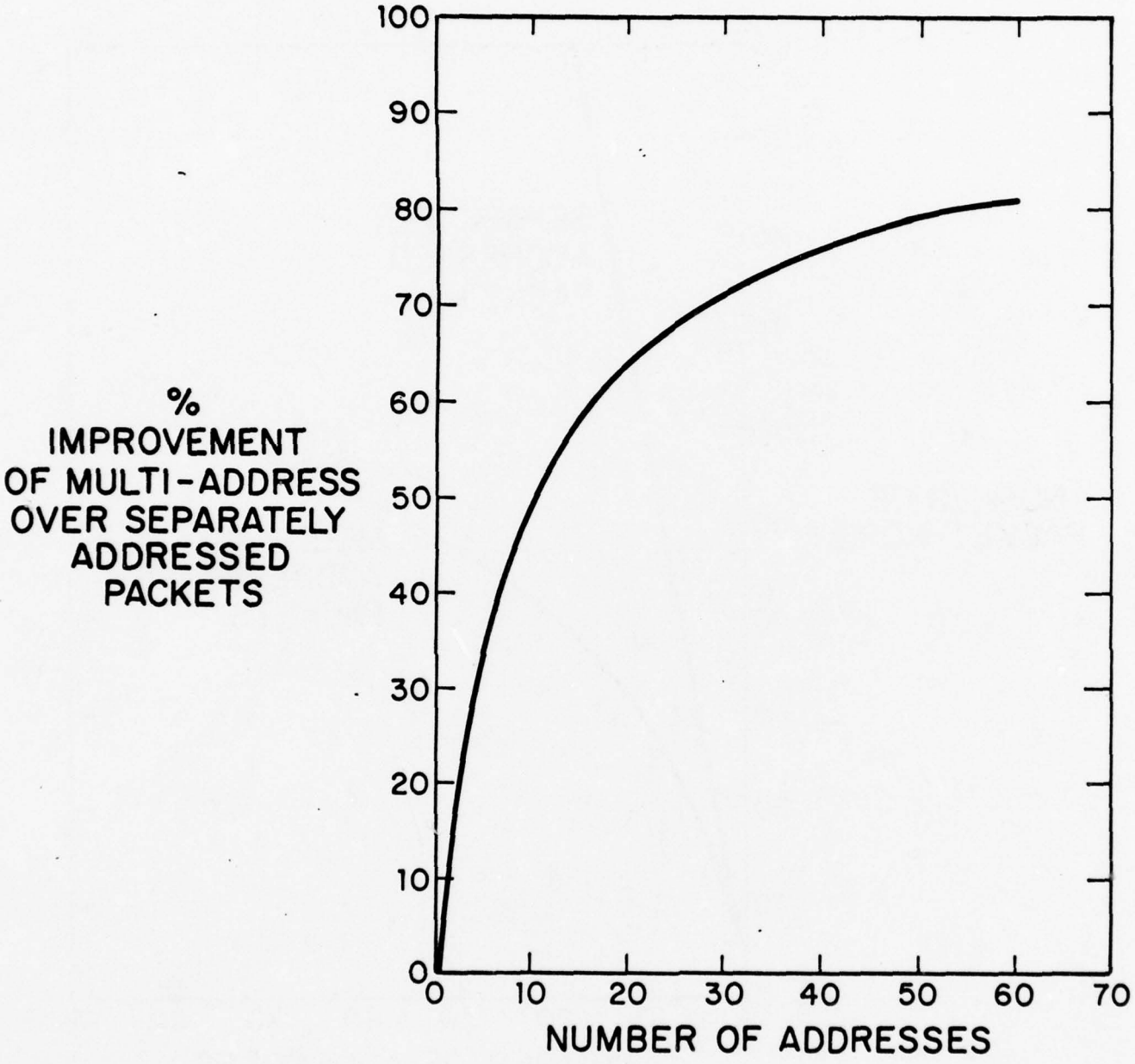


Figure 7-2 Multi-Address Packets: Percentage Improvement

In General	For ARPANET
$a=1 \quad 0$	$a=1 \quad 0$
$a=n-1 \quad \frac{1}{2}\ \frac{h-1}{h}$	$a=8 \quad 40\%$ (from 44 to 26)
$a=n-1 \quad \frac{h-1}{h}$	$a=61 \quad 81\%$ (from 336 to 61)

% Improvement

For the ARPANET,

number of nodes n = 62
connectivity c = 2.5
"depth" d = 6.36
path length h = subtree size s = 5.5
trees/arc = $\frac{n-1}{c}$ = 25.2

Thus, in a network with a path length of 5, the greatest improvement in number of packet hops, 80%, occurs when addressing all other nodes in the network. When addressing a number of destinations equal to the square root of the number of nodes in the network, half of this relative improvement, or about 40% is obtained. We have calculated for the ARPANET that addressing as few as 5 to 10 destinations in the same packet results in a savings of 25% to 50% of the packet hops required with separately-addressed packets.

The issues of formatting packets and messages with logical addresses and multi-destination addresses deserve some consideration. Group addressing is simpler to implement in the

network since it requires a relatively small change to the subscriber software. On the other hand, multi-destination addressing is more flexible and useful to the subscribers but requires a fairly major change in the subscriber-to-network format. Careful attention must also be given to the interaction between logical addressing and group addressing, since group addressing in general should permit reference to logical as well as physical addresses. If a group address refers to several logical addresses as well as physical addresses, then the translation of logical to physical addresses must take place at the source node. Figures 7-3, 7-4, and 7-5 present the packet formats we propose for multi-address and group address packets in the ARPANET.

There are considerable advantages to installing the enhanced message addressing modes discussed in this paper. Logical addressing provides for considerable operational flexibility and reliability. The use of multi-destination and group addressing has been shown to lead to significant reduction in network traffic, even for the case of relatively few destinations per message. One of the important conclusions from this work is that datagram networks which facilitate the use of logical addressing and multi-destination addressing may have some important advantages over virtual circuit networks. These advantages have not yet been fully considered by the network design community.

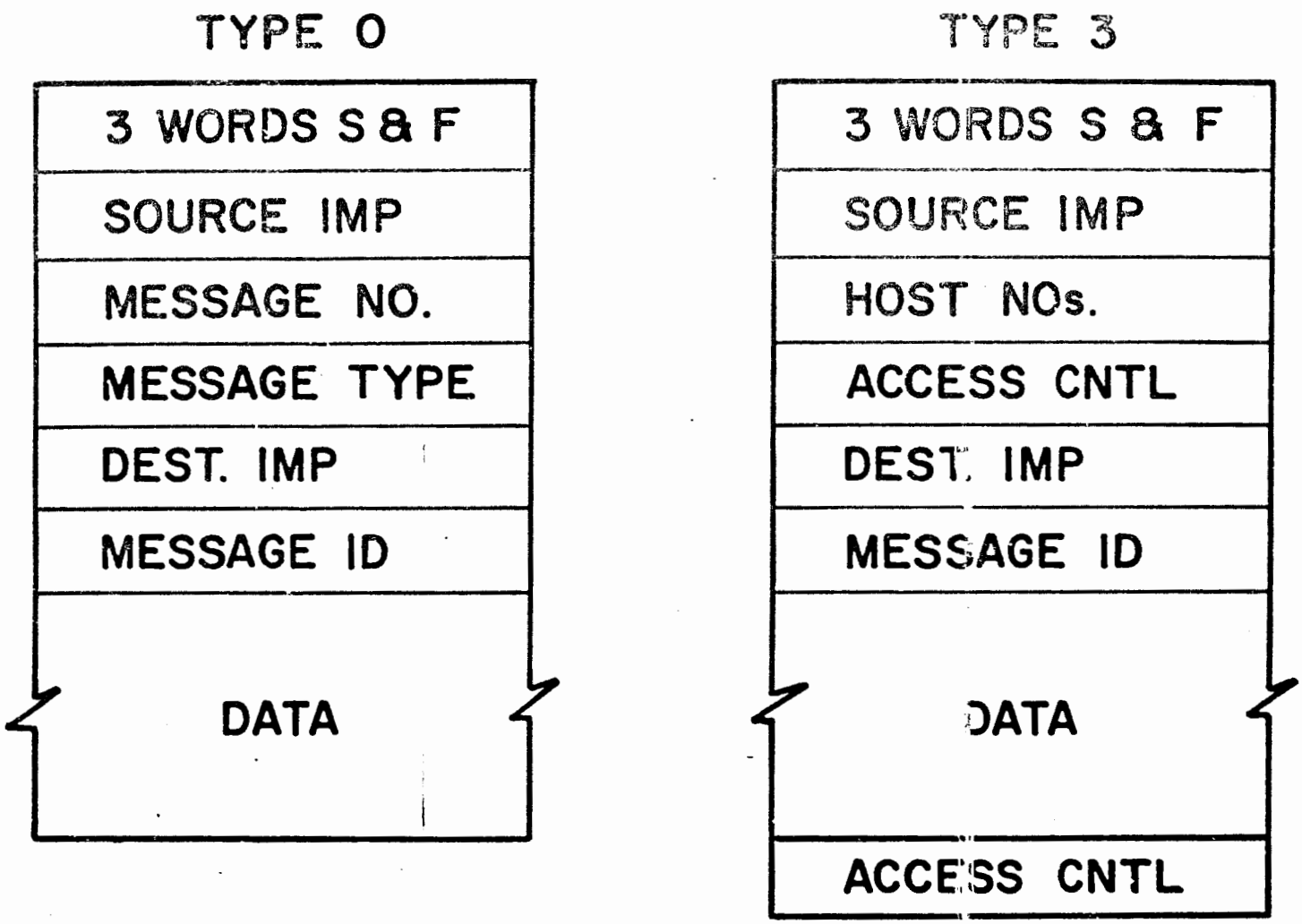


Figure 7-3 Present Packet Formats

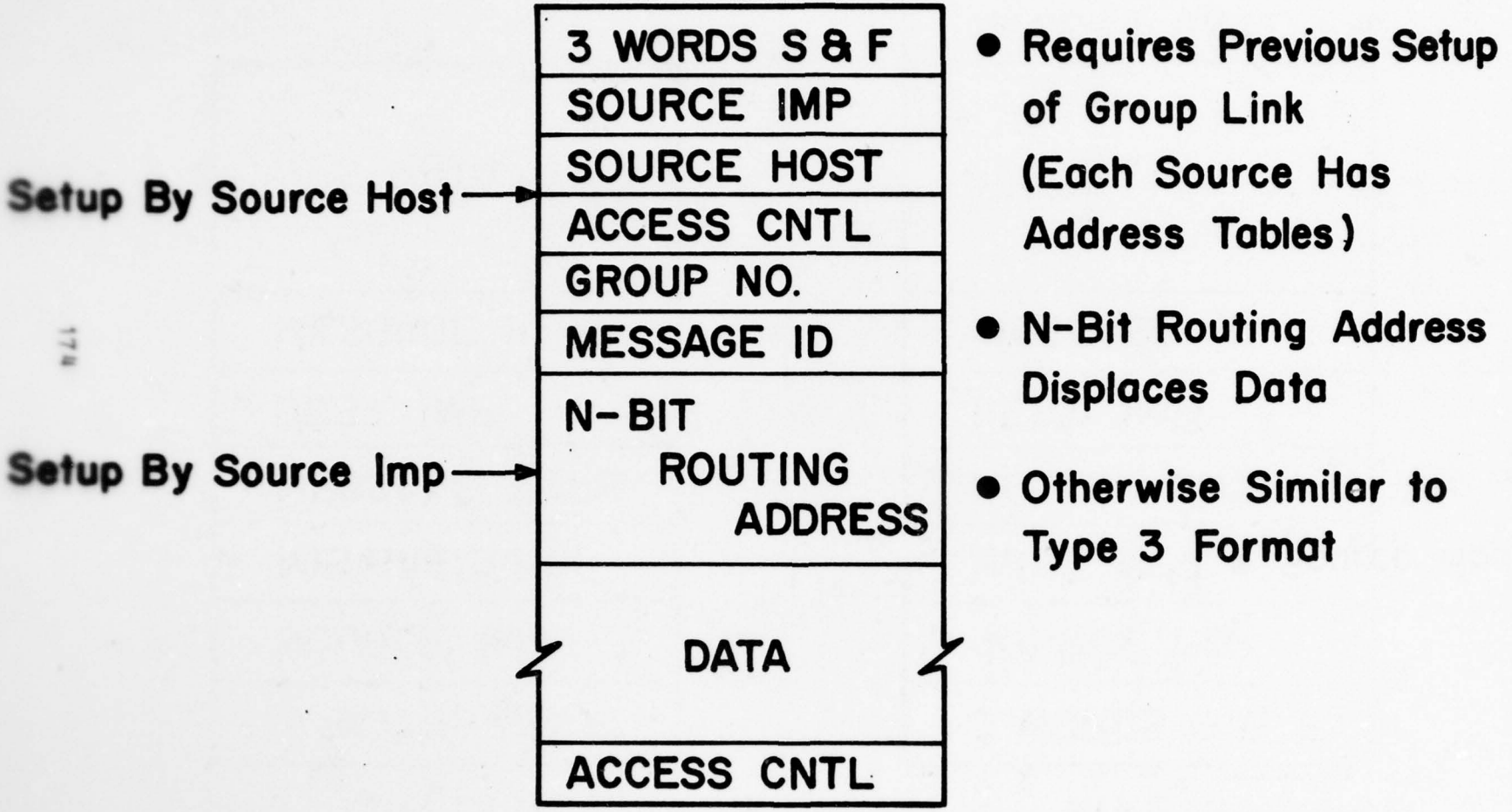


Figure 7-4 Group Address Packet Format

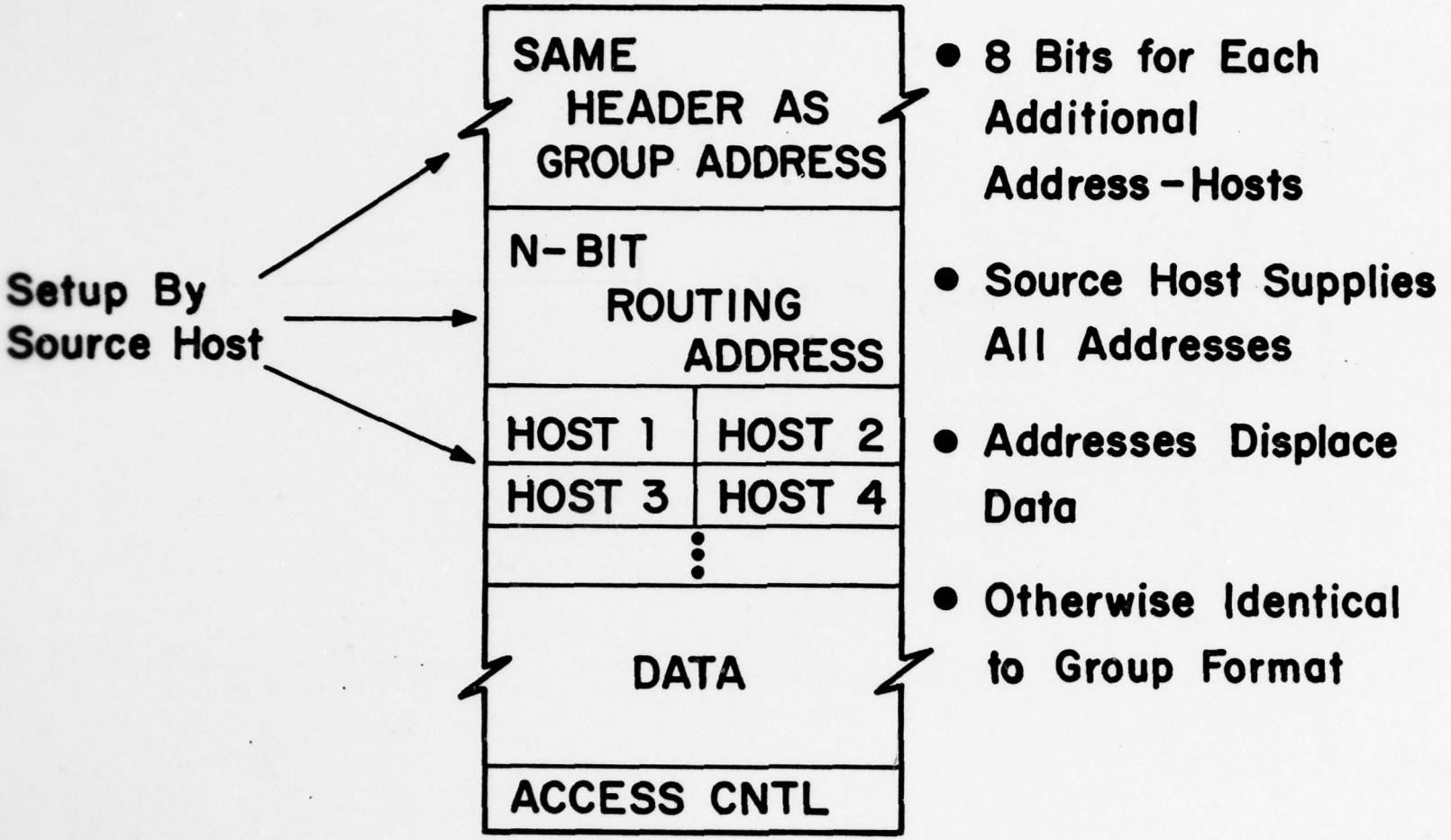


Figure 7-5 Multi-Address Packet Format

REFERENCES

[1] D.B. Johnson, "Efficient Algorithms for Shortest Paths in Sparse Networks," J. ACM, Vol. 24, pp. 1-13; January 1977.

[2] E.W. Dijkstra, "A Note on Two Problems in Connection with Graphs," Numer. Math, Vol. 1, pp. 269- 271; 1959.

Appendix 1 DATA GATHERING METHODOLOGY

Our basic technique for gathering data was to use the IMP's packet-trace package. For every nth packet which arrives at a given IMP, where n is a settable parameter, packet-trace generates a trace block, which contains certain information about the packet. Periodically, the IMP gathers together all the trace blocks, stuffs them into a packet, and sends the packet to a specified collection point. This enables us to gather information about a sample of the packets which pass through a given IMP.

The trace block contains the following information about the packet it corresponds to:

a. The time the packet arrived in the IMP.

b. The time the packet was queued for transmission.

c. The time the first bit of the packet was transmitted.

d. The source and destination of the packet.

e. The output line (or host, as the case may be) on which the packet was transmitted.

f. The priority of the packet.

g. The number of data words in the packet.

h. The number of packets which are ahead of the given packet on the modem transmission queue. (This is not part of the standard trace package--it had to be patched in.)

To compute the processing delay (delay due to queueing for the processor as well as to the processing itself), we subtracted a from b. The modem queueing delay is obtained by subtracting b from c. The transmission delay is a function of g. The propagation delay is a constant for a given line. Thus the trace blocks contain enough information to enable us to compute each of the four components of delay.

There are, however, certain important limitations to this methodology. These limitations have to do with sampling frequency, sampling content, timing considerations, and the natural variability in the traffic itself.

a. Sampling Frequency. We originally attempted to time every 5th packet through an IMP. We found, however, that we were unable to collect the resultant trace output on TENEX, perhaps because of TENEX throughput limitations. When we attempted to trace every 10th packet, we had no collection problems, so we gathered all our data at the frequency of every 10th packet. The less frequently the data is sampled, the more likely it is that certain characteristics of the data are merely artifacts of the sampling technique. Our data must always be evaluated with this fact in mind.

b. Sampling content. The first time we looked at the trace data, we noticed that packets which originated from local hosts had processing delays approaching a quarter of a second. Since tandem packets had much smaller processing delays, we concluded that most of the delay experienced by the host packets was due to end-end considerations. As we were only interested in measuring the delay due to store-and-forward considerations, we were forced to exclude from our sample all packets which originated from local hosts. We could not prevent trace blocks from being generated for such packets--rather, we had to ignore those trace blocks. In addition to being arbitrary, this further reduced our effective sampling frequency.

c. Timing considerations. The time stamps in the trace block are taken from the 100-microsecond clock. This clock wraps around every 6.6 seconds. This is not a serious problem in determining the delays experienced by a particular packet since it may be safely assumed that no packet stays on any queue for more than 6.6 seconds. The wrap-around is a problem though in trying to determine the relative arrival times of different packets. For the purpose of plotting delay vs. time, we assumed that the clock never wrapped around more than once between the creation of successive trace blocks. This assumption may or may not be true in all cases.

d. Vaiability in traffic. User-created traffic is extremely variable, and when comparing different measurements, we have no way of controlling for it. We gathered data under three different conditions--ordinary traffic, artificially created heavy load, and artificially created heavy load with reduced-rate routing.

a. Ordinary traffic. We gathered data from four IMPs--LINC, ACCAT, ISI22, and MIT6. Data from MIT6 was gathered on two separate occasions, one of which was the same day that data was gathered from the other three IMPs. In all cases, we traced every tenth packet for a period of about 10 minutes.

b. Artificially created heavy load. We traced every 10th packet in MIT6 for a period of about 15 minutes. Five minutes into the run we turned on WPAFB's message generator, having it send single packet messages to MIT44 at the maximum frequency. We turned the message generator off after five minutes. We did this twice, once with minimum size packets and one with maximum size packets.

c. Artificially created heavy load with reduced-rate routing. We repeated the previous experiment with the following differences:

i. WPAFB, MIT6, and MIT44 were patched to always send routing at the minimum frequency (once per slow tick). MIT6's third neighbor, CCA, was not patched, and presumably continued to send routing at the constant Pluribus frequency of twice per slow tick.

ii. Instead of using minimum size single packet messages for one of the runs, we used maximum size eight-packet messages.

APPENDIX 2. A COMPLEXITY BOUND FOR THE INCREMENTAL SHORTEST PATH PROBLEM

Summary. The average number of nodes in the subtree of a given line in a tree is equal to the average path length from the root to any node. This number divided by the average number of lines per node represents an upper bound on the expected number of route changes necessary at each node when recomputing shortest paths to all other nodes after the distance value for one line is changed.

A2-1 Introduction

We will consider a network with N nodes and L lines, each line having a non-negative distance. Lines are considered to be directed arcs and each pair of adjacent nodes is connected by one line in each direction. We are concerned with the calculation of shortest paths from a source node to all other nodes, given that the distance of one line in one direction has changed. To this end, we will assume that the shortest path tree from the source node to all other nodes has been previously computed, and that incremental changes only are required.

A2-2 Path Length and Subtree Size

Consider the shortest path tree at the source node. We begin with some definitions and a result which holds for any tree:

Definition: The path length h_i, to node i is the number of intermediate lines on the path from the source to node i in the tree.

Definition: A node j is a descendant of a node i if the shortest path from the source to j includes shortest path from the source to j. (This implies $h_j \geq h_i$).

Definition: The subtree size, s_i, of node i is the number of nodes which are descendants of node i, including node i.

let

$$h = \frac{1}{N-1} \sum_{i=1}^{N} h_i \quad \text{average path length in the tree}$$

$$s = \frac{1}{N-1} \sum_{\substack{i=1 \\ i/\text{source}}}^{N} s_i \quad \text{average (proper) subtree size in the tree}$$

Theorem 1. In any tree, s=h

Proof: Note that h_i is the number of subtrees in which node i appears. Therefore

$$s = \frac{1}{n-1} \sum_{i=1}^{n} h_i = h$$

Q.E.D.

This surprising result can be shown in other ways including the following:

let a_i = the number of nodes with path length i

d = the maximum path length in the tree

b_i = the average subtree size of all nodes with path length i;

$$b_i = \frac{1}{a_i} \sum_{j=i}^{d} a_j$$

Then we have that

$$h = \frac{1}{N-1} \sum_{i=1}^{d} i(a_i)$$

$$s = \frac{1}{N-1} \sum_{i=1}^{d} b_i(a_i)$$

$$= \frac{1}{N-1} \sum_{i=1}^{d} \sum_{j=i}^{d} a_j$$

$$= h \qquad \text{Q.E.D.}$$

A2-3. The Incremental Shortest Path Problem

We next examine the effect on the shortest path tree of changes to the network. Let

c = average node degree in the network; $c = \frac{L}{N}$

Definition: A network change refers to the addition or deletion of a single network line or to the change of the distance value associated with some line.

Definition: A routing change for node i is required when, as a result of a network change, shortest distance route from the source to node i changes.

Theorem 2. When a line is deleted, the expected number of routing changes per source node is equal to h/c.

The probability that a given line is in the shortest path tree at a given source is equal to the number of lines in the tree divided by the number of network lines which can be in the tree:

$$\frac{N-1}{cN-c} = \frac{1}{c}$$

If the deleted line from node x to node y was in the tree, then node y and its subtree all require routing changes. This is, on average, a group of h nodes. If the deleted line is not in the tree, no routing changes are needed. Therefore, the expected number of routing changes is given by

$$h\left(\frac{1}{c}\right) + 0\left(\frac{c-1}{c}\right) = \frac{h}{c} \qquad \text{Q.E.D.}$$

Finally, we can generalize the result of Theorem 2 by considering all types of network change.

We make the following assumptions:

1. The distance of all lines is the same, which minimizes h for the starting network.

2. Only single network changes from the starting network are considered.

3. All network changes are equally likely; both the line affected and the size of the change are chosen randomly.

Theorem 3: The expected number of routing changes per source node is bounded by h/c for any network change.

Proof: There are four cases to consider: the addition or deletion of a line, and the increase or decrease of the distance of a line. Let r_l (i,j) = number of routing changes required at a source node when the distance of line l changes from i to j.

Let K = maximum distance/line (assume distances are integers)

m = distance value used for a deleted line; $m \geq K(L)$

Then we can write

R_1 = expected number of routing changes for line deletion

$$R_1 = \frac{1}{L} \sum_{l=1}^{L} \frac{1}{K} \sum_{i=1}^{K} r_1(i,m)$$

R_1 = h/c by Theorem 2

R_2 = expected number of routing changes for line addition

$$R_2 = \frac{1}{L} \sum_{l=1}^{L} \frac{1}{K} \sum_{i=1}^{K} r_1(m,i)$$

R_3 = expected number of routing changes for a distance increase

$$r_3 = \frac{1}{L} \sum_{l=1}^{L} \frac{1}{K} \sum_{i=1}^{K} \frac{1}{K-i} \sum_{x=1}^{K-1} r_1(i,i+x)$$

R_4 = expected numbers of routing changes for a distance decrease

$$R_4 = \frac{1}{L} \sum_{l=1}^{L} \frac{1}{K} \sum_{i=1}^{K} \frac{1}{i-1} \sum_{x=1}^{i-1} r_1(i,i-x)$$

To prove the theorem we must show

$$R_2 \leq R_1$$
$$R_3 \leq R_1$$
$$R_4 \leq R_1$$

For any given network change, $r_1(i,i+x) \geq r_1(i+x,i)$, since all nodes which are affected by the increase in distance to i+x will also be affected by an equal decrease back to i, except for those which have an equal-distance path and do not require a second routing change. Therefore,

$$r_1(m,i) \leq r_1(i,m), \text{ and thus}$$

$$R_2 \leq R_1$$

For any given network change and any value of $y > 0$, $r_1(i, i+x+y) \geq r_1(i,i+x)$. This is, the bigger the network change, the more routing changes that are required. Therefore,

$$\frac{1}{K-i} \sum_{x=1}^{K-i} r_1(i,i+x) \leq r_1(i,m); \text{ and thus}$$

$$R_3 \leq R_1$$

Similarly, we can show

$$R_4 \leq R_2$$

A2-4. Numerical Values

It is relatively simple to develop a lower bound for the value of h in a regular network (equal degree c for all nodes). The lower bound is obtained when the shortest path tree is full and balanced. In the terminology used in Theorem 1, $a_1 = c$, $a_2 = c(c-1)$, and in general

$$a_i = c(c-1)^{i-1}$$

Then, $$N-1 = \sum_{i=1}^{d} a_i = \frac{c((c-1)^d-1)}{c-2}$$

Therefore, $$h = d \frac{(c-1)^d}{(c-1)^d-1} - \frac{1}{c-2}$$

Numerical values for h/c are plotted in Figure 1 for c = 2.3, and 4. These curves show that the expected number of routing changes per node per routing change remains very small, in the range of 1 to 5, even for networks with 10,000 nodes.

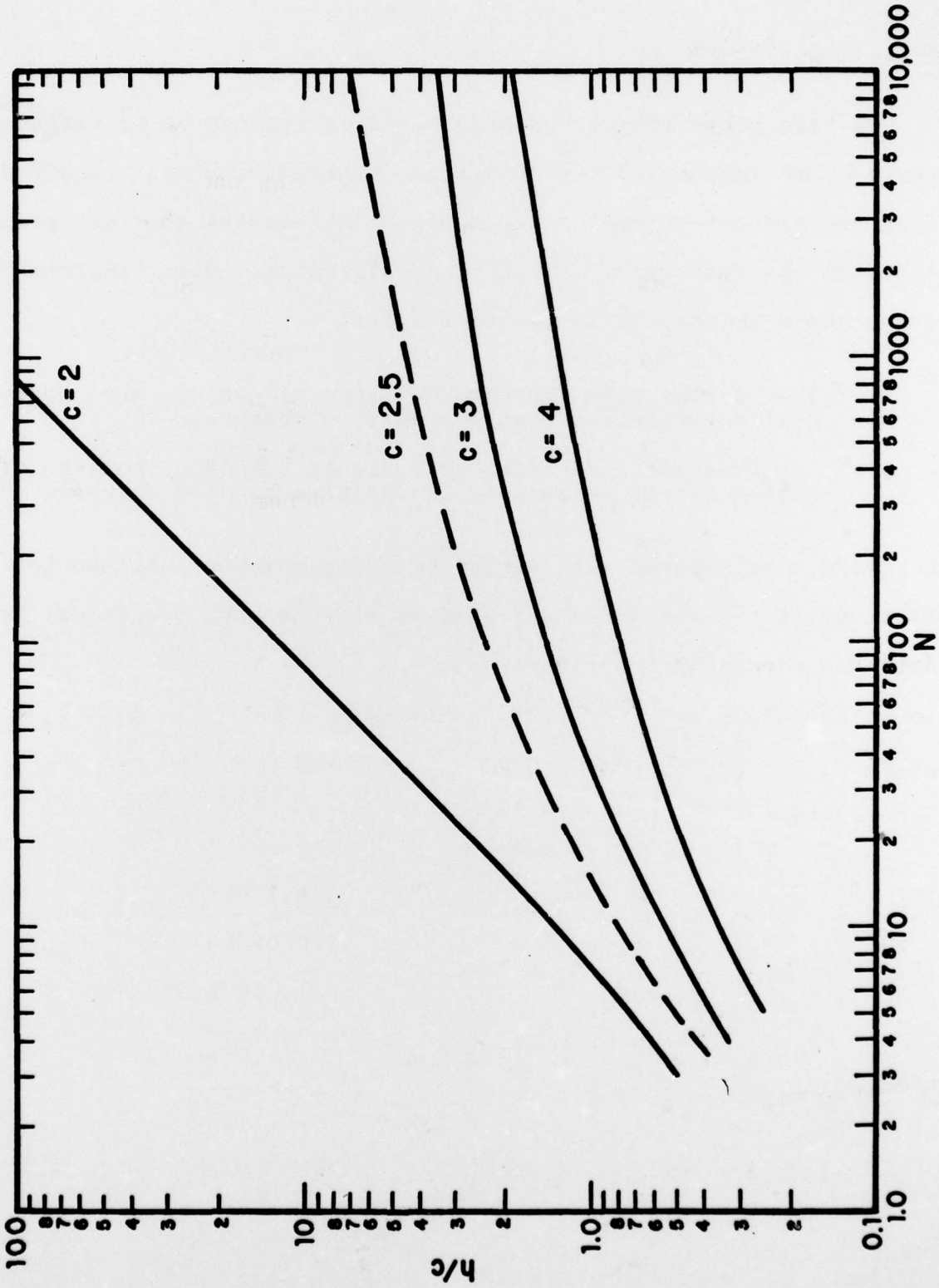


Figure A2-1 Lower Bound for h/c

A2-5. Conclusions

We have shown some surprisingly simple relationships between average path length and the number of routing changes required after a network change, given that an incremental shortest path algorithm is employed. Such an algorithm has important advantages with respect to its complexity:

-- The running time depends on h and c, which are basic design parameters that are easy to measure.

-- The time per node grows very slowly with the number of nodes, and is relatively insensitive to node degree.

Therefore, an incremental algorithm which computes new shortest paths only for those nodes affected by each network change may be desirable for certain applications.